WL-TR-96-3126

THE APG-70 RADAR SIMULATION MODEL FINAL REPORT

Lucy Garcia
Dave Gehl
Chris Buell
John Hassoun

Veda, Incorporated
5200 Springfield Pike, Suite 200
Dayton, OH 45431-1255

September 1996

Final Report For Period September 1994 to March 1996

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED

FLIGHT DYNAMICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7562

NOTICE

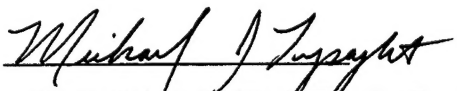
When government drawings, specifications, or other data are used for any purpose other than in connection with a definitely government-related procurement, the United States Government incurs no responsibility nor any obligation whatsoever. The fact that the government may have formulated, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise in any manner construed, as licensing the holder or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related there to.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

GREGORY J. BARBATO
ENGINEERING PSYCHOLOGIST
COCKPIT DEVELOPMENT SECTION
ADVANCED COCKPITS BRANCH

RICHARD W. MOSS
CHIEF, COCKPIT DEVELOPMENT SECTION
ADVANCED COCKPITS BRANCH
FLIGHT CONTROL DIVISION



MICHAEL J. LYSAGHT, LTCOL, USAF
CHIEF, ADVANCED COCKPITS BRANCH
FLIGHT CONTROL DIVISION
FLIGHT DYNAMICS DIRECTORATE

DAVID P. LEMASTER
CHIEF, FLIGHT CONTROL DIVISION
FLIGHT DYNAMICS DIRECTORATE
WRIGHT LABORATORY

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify Wright Laboratory; Flight Dynamics Directorate; WL/FIPA Bldg 146; 2210 Eighth Street Ste 1; Wright-Patterson Air Force Base, OH 45433-7511 USA

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 96	3. REPORT TYPE AND DATES COVERED Final Report (09/21/94 - 03/31/96)

4. TITLE AND SUBTITLE
THE APG-70 RADAR SIMULATION MODEL FINAL REPORT

5. FUNDING NUMBERS
F33615-93-D-3800, Delivery Order 00013
PE: 62201 F
PR: 2403
TA: 04
WU: SP

6. AUTHOR(S)
Lucy Garcia, Dave Gehl, Chris Buell, John Hassoun

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Veda Incorporated
5200 Springfield Pike, Suite 200,
Dayton, OH 45431-1255

8. PERFORMING ORGANIZATION REPORT NUMBER
63351-96U/P61213

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Cockpits Branch
WL/FIGP, Bldg. 146
221 8th Street, Suite 1
Wright-Patterson AFB, OH 45433-1255

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-96- 3126

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
UL

13. ABSTRACT (*Maximum 200 words*)

This report describes contract activities that were completed by Veda Incorporated and Hughes Training Incorporated from September 1994 through March1996 under DO 0013 of the Pilot Factors Contract F33615-93-D-3800. Specifically, this report summarizes the long term requirements that drove the development of an APG-70 radar capability and the PVI of the radar, as integrated into the CSIL simulation facility. This report also summarizes the design considerations, design approach, design implementation, and the current the current status of the overall software development. In addition, recommendations are identified within the report to further the integration and fidelity of the APG-70 radar simulation within CSIL.

The sections devoted to the software design architecture and implementation are provided so that a proficient software engineer will be able to thoroughly understand the software design and the implementation without having to review the source code.

14. SUBJECT TERMS
Common Cockpit, Control and Displays, Pilot Vehicle Interface, Transport/Tanker Aircraft, Radar, Radar Models, APG-70 Simulation

15. NUMBER OF PAGES
173

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS, TERMS, AND ABBREVIATIONS

A/A	Air-To-Air
A/G	Air-To-Ground
AGL	Above Ground Level
AHRS	Attitude Heading Reference System
ARMT	Armament
AZ	Azimuth
BCN	Beacon
BRT	Bright
CHAN	Channel
CONT	Contrast
CSIL	Cockpit Integration Division's Crew Systems Integration Laboratory
DCL	Decluttter
DMA	Defense Mapping Agency
DO	Delivery Order
DoD	Department of Defense
DTED	Digital Terrain Elevation Data
EO	Electro-Optical
FLIR	Forward Looking InfraRed
GC	Gain Control
GUF	Grand Unified File-Format
HMD	Helmet-Mounted Display
HOTAS	Hands-On Throttle and Stick
HRM	High Resolution Map
HSI	Horizontal Situation Indicator
HTML	Hyper-Text Markup Language
HUD	Head Up Display
IMPACT	Integrated Mission Precision Attack Cockpit
INS	Inertial Navigation System

INV	Invalid
IP	Initial Point
IPVU	Interleaved Precision Velocity Update
IR	Infra-Red
LANTIRN	Low Altitude Navigation and Targeting Infra-Red for Night
LIFO	Last In/First Out
ML	Multi-Look
MN	Mission Navigator
MSL	Mean Sea Level
NM	Nautical Miles
PPI	Plan Position Indicator
PROG	Program
PSL	Pattern Steering Line
PVI	Pilot-Vehicle Interface
PVU	Precision Velocity Update
R/AZ	Range/Azimuth
RBM	Real Beam Mode
RCD	Record
RDR	Radar
RS	Radar System
SAR	Synthetic Aperture Radar
SEC	Second
SNIFF	SNIFF Mode-Receive Only
STO	Store
TAF	Tactical Air Forces
TDC	Target Designation Controller
TF	Terrain Following
TGT	Target
TO	Technical Order
TR	Technical Report

TSD	Tactical Situation Display
UPDT	Update
VTR	Video Tape Recorder
WPN	Weapon
3-D	Three-Dimensional

1. INTRODUCTION AND SUMMARY

The APG-70 radar simulation model will be used by the Integrated Mission Precision Attack Cockpit Technologies (IMPACT) program of the Advanced Cockpits Branch of the Wright Laboratory (WL/FIGP) while developing concepts that will enable pilots to perform missions in single-seat aircraft. Items that were developed under Delivery Order (DO) 0013 are (1) the software source code, that simulates the real beam map (RBM) and the high resolution map (HRM, specifically the patch map) modes of the F-15E's air-to-ground radar system, (2) the respective Hyper-Text Markup Language (HTML) files that provide the electronic documentation of the source code, and (3) this APG-70 Radar Simulation Model Final Report.

1.1 Background

The IMPACT program is a research and development effort funded by the Wright Laboratory at Wright-Patterson Air Force Base. The objective of the program is to analyze, design, develop, and test cockpit controls and display concepts that will enable pilots to perform a precision strike mission, against multiple mobile and fixed targets, at night and in adverse weather, in a single-seat aircraft. The IMPACT program was conceived in response to the following Tactical Air Force (TAF) mission need statement (TAF 401-91):

"Current aircraft/weapon systems lack the capability to accomplish key objectives against surface targets under certain adverse weather conditions. . . .Current Department of Defense (DOD) precision strike systems, using laser, electro-optical, or infra-red guidance, demand virtually clear line-of-sight from designating aircraft or the data link weapon to the target. . . .Consequently, common environmental conditions such as low-to-mid-altitude clouds, various forms of precipitation, and phenomena limiting visibility significantly restrict our ability to strike fixed, re-locatable, and moving targets. . . .Enhanced ability to precisely attack fixed, re-locatable, or moving land and maritime targets under adverse environmental conditions is required in the near term."

The cockpit analysis and design approach of the IMPACT program rely on the F-15E as a baseline cockpit for developing concepts that will decrease high workload areas for the pilot and enhance the pilot's ability to perform missions in single-seat aircraft. One known high workload area for a single crew member is the operation of the radar and the interpretation of the radar's real-time images (Montecalvo, A.J., et al. 1994). While performing the radar-related activities, the pilot must also perform the flying, offensive, and defensive activities. To accurately assess the extent of the workload, resulting from a single crew member performing all activities typically handled by two crew members, subject pilots must be exposed to representative functions and tasks that are accomplished during critical mission segments. Therefore, the IMPACT program is tasked with simulating the baseline (F-15E) cockpit's APG-70 radar in its in-house facility, the Crew Systems Integration Laboratory (CSIL). Once the level of crew task difficulty is identified, the IMPACT program will provide candidate solutions for reducing crew member workload either by simplifying or automating radar-related activities, or by simplifying or automating other activities that occur during the high-workload segments of the mission.

1.2 Objectives of the APG-70 Radar Development

To accurately assess pilot workload and various task allocation schemes during the F-15E cockpit development efforts, it became apparent that a realistic representation of the baseline aircraft's radar system would be required. A realistic representation of the F-15E's APG-70 radar would provide the ability to completely analyze the task loading dynamics of a single pilot performing radar functions during the attack phase of a mission. Three objectives were identified: to develop a visual radar simulation model; to develop a flexible radar simulation model; and to comply with CSIL hardware and software constraints.

Objective One—Develop a Visual Radar Simulation Model

The *first* objective was to simulate the APG-70 radar PVI mechanization like the F-15E's mechanization. In particular, a representative real beam radar model and a synthetic aperture radar model were identified to be simulated in a real-time pilot-in-the-loop simulation environment. The simulation would ensure that the performance measures taken in subsequent scientific studies would include the full breadth of pilot tasks associated with

detecting, recognizing, and designating fixed and relocatable targets in a representative mission scenario.

Objective Two—Develop a Flexible Radar Simulation Model

Once the decision was made to simulate the APG-70 radar, additional requirements were identified. During in-house group meetings, the IMPACT team determined that a *second* objective would be to integrate the radar simulation into CSIL so that it not only represented the F-15E pilot-vehicle interface (PVI), but that the radar simulation would also be flexible to evaluate advanced design concepts. For example, the technology assessment conducted during the IMPACT mission analysis and interface requirements definition efforts identified sensor fusion and an electronically steerable array radar as advanced radar concepts that show promise for enhancing target detection, recognition, and designation. However, the PVI designs of these advanced radar concepts must be optimized and validated through pilot-in-the-loop simulations. Instead of purchasing a simulation of the F-15E APG-70 radar, the decision was made to develop the software in-house so that the source code would be available and modifiable. By designing flexibility and rapid reconfiguration into the radar software through object-oriented design, the ability to modify the F-15E simulation would be enabled. With the ownership of the source code, enhancements and advanced radar concepts could be mechanized into the CSIL facility to perform subsequent validation and verification studies involving state-of-the-art technologies. In addition, the radar simulation source code would be available for distribution to other similar USAF laboratories (e.g., Crew Station Evaluation Facility) taking advantage of software code reuse and the object-oriented design.

Objective Three—Comply with CSIL Hardware and Software Constraints

Various hardware and software constraints were also identified through the in-house group meetings. Since the radar simulation would be integrated into CSIL, using the existing computer hardware that composes the IMPACT cockpit simulation would be required for hosting the radar simulation. Therefore, the *third* objective was to use the existing computer hardware and the corresponding software environment to integrate the APG-70 radar simulation into the CSIL. The reconfigurable cockpit simulator contains a BARCO Retrographics 801 that combines a digital chassis and a self-contained projection cabinet to

render high resolution images generated by a Silicon Graphics Onyx workstation, which is configured with Reality Engine graphics hardware. To promote ease of portability and integration into the CSIL facility, the only tools used in the development of the F-15E APG-70 radar simulation would be the tools employed by the CSIL facility, including the object-oriented language entitled C++ and the respective Iris development environment, the Iris Operating System, the Iris Performer Libraries, and the Iris Graphics Libraries. The MultiGen visual database modeling system would also be employed in the development of the F-15E APG-70 radar simulation. For accuracy and completeness, the real beam and synthetic aperture images produced by the APG-70 radar model must correlate to the visual scenes rendered in the current IMPACT simulation by the Iris Performer visual modeling system.

1.3 Scope of Report

This report describes contract activities that were completed by Veda Incorporated and Hughes-Training Incorporated from September 1994 through March 1996 under DO 0013 of the Pilot Factors Contract F33615-93-D-3800. Specifically, this report summarizes the long term requirements that drove the development of an APG-70 radar capability and the PVI of the radar, as integrated into the CSIL simulation facility. This report also summarizes the design considerations, design approach, design implementation, and the current status of the overall software development. In addition, recommendations are identified to further the integration and fidelity of the APG-70 radar simulation within CSIL.

The sections devoted to the software design architecture and implementation (Sections 3 through 6) are provided so that a proficient software engineer will be able to thoroughly understand the software design and the implementation without having to review the source code.

1.4 Executive Summary

The design and development of the APG-70 radar simulation model is based on a structured systems engineering process, which included a long term requirements definition, software design, systems integration, testing, and documentation.

The functional level requirements of the radar model stem from the IMPACT project's prerequisites to simulate the RBM plan position indicator (PPI) and the High Resolution Map (HRM) patch map modes of the F-15E APG-70 radar. Other modes, such as the complete HRM PPI, the precision velocity update, the air-to-ground beacon, and the air-to-air were considered potential growth options, but were not developed under DO 0013.

A Silicon Graphics Reality Engine or an Onyx Workstation were defined as the computer hardware requirements for the APG-70 radar simulation. The software architecture of the simulated APG-70 radar model requires the input of digital cultural and terrain data from the Defense Mapping Agency (DMA) databases, uses MultiGen to process the data, and feeds the data into the radar model, which calculates and displays radar image. For correctness and completeness, the radar image rendered by the APG-70 radar simulation must correlate with the visual scene terrain database as rendered in the real-time simulation by Iris Performer.

Upon completing the design and development of the simulation software, the RBM PPI and the HRM patch map modes were integrated into the CSIL facility. The RBM PPI mode is used to identify gross terrain features for navigation, weather detection, and to cue the radar to a specific point for high resolution patch mapping. The HRM patch map mode is used to cue electro-optical (EO) and infra-red (IR) sensors to a specific point on the ground, to perform position updates, and to designate in-the-weather targets.

Both modes, the RBM PPI and the HRM patch map, of the APG-70 radar system have different levels of integration into the CSIL facility. The RBM PPI mode is integrated into the CSIL facility with respect to the IMPACT cockpit. This mode can be accessed within the simulation with overlaying display formats decorating the sweeping of the RBM mode display output. The digital elevation data files are loaded in memory for the gross terrain features of the RBM PPI mode of the APG-70 radar simulation. In real-time, the position of the aircraft via the avionics aerodynamic model drives the APG-70 radar simulation's output. An input monitoring program monitors keyboard input to control changes within the APG-70 radar model, such as radar range selection, antenna elevation and azimuth, and scan width selection. Since the APG-70 radar produces high range and azimuth resolutions using

synthetic aperture radar (mapping) techniques within the HRM PPI and patch map modes, the terms HRM and SAR are used interchangeably throughout this report. The HRM patch map mode has not been integrated as fully as the RBM PPI mode due to time constraints. The HRM patch map mode uses the same MultiGen flight file created under the IMPACT project, which drives the visual scene for the IMPACT simulation and the RBM PPI mode uses the corresponding IMPACT digital elevation data files. The MultiGen loader is used to store the database flight files in memory for the HRM patch map mode of the APG-70 radar simulation. Short and long term recommendations are identified within the report to further the integration and fidelity of the APG-70 radar simulation within CSIL.

2. RADAR REQUIREMENTS DEFINITION

The Integrated Mission Precision Attack Cockpit Technologies (IMPACT) team thoroughly reviewed the APG-70 radar mechanization in the F-15E and prioritized the various operating modes. The review was performed to bound the scope of the radar development effort and to select only those modes that were required to conduct pilot-in-the-loop simulations. The two APG-70 radar modes that had the highest priority were the real beam map (RBM) and the high resolution map (HRM) modes. The IMPACT team determined that the precision velocity update, beacon map, air-to-ground ranging, and air-to-air modes were not required, but should be considered during the radar development as growth items for future implementation.

The information that follows provides an overview of the actual APG-70 radar within the F-15E aircraft and a brief description of the APG-70 pilot-vehicle interface (PVI), including the RBM plan position indicator (PPI) and the HRM patch map modes that were developed under Delivery Order (DO) 0013.

2.1 The F-15E APG-70 System Overview

The APG-70 radar system, currently employed on the F-15E aircraft, provides the capability to assist the crew in navigating, detecting, and designating ground targets. The radar has multiple modes of operation, but the two primary modes are the RBM and the HRM modes. The RBM mode's main functions are to identify gross terrain features for navigation, to detect weather conditions, and to cue the HRM. The HRM's main functions are to perform wide area searches, to update position, to perform electro-optical (EO) sensor cueing, and to designate in the weather targets. Both modes were identified by the IMPACT team as the primary modes of interest for the implementation of the APG-70 radar simulation into the Crew Systems Integration Laboratory (CSIL) facility.

Also included within the F-15E aircraft, the APG-70 radar system's precision velocity update (PVU) mode provides the precision navigation and also cues the HRM formation. The air-to-ground beacon map mode of the APG-70 radar system provides the navigation update

and offset bombing ability. The air-to-ground ranging mode of the APG-70 radar system provides visual weapon delivery; however, it is a mode not selectable from the air-to-ground radar display format.

Within the F-15E aircraft the radar functions can be controlled via multi-functional displays (MFD), radar controls on cockpit control panels, and the hands-on-throttle-and-stick (HOTAS). These controls and displays provide the interface for the full gamut of radar functions. In the RBM mode, the crew has the capability to quickly select the HRM mode for high resolution mapping. The range of radar coverage is also selectable and includes 4.7, 10, 20, 40, 80, and 160 nautical miles (NM). The radar is capable of accessing stored sequence point (pre-planned geographical position) data and of displaying the data's location relative to the aircraft's current position. The radar also has a "receive only" mode that can be used to detect the jamming of radar channels.

Within the RBM mode, the radar operates in one of five frequency bands and over eight channels per band, which can be selected via the MFD in the F-15E cockpit. The azimuth and elevation of the radar beam can be controlled, within limits, by the crew so that an optimal radar image can be obtained. A radar cursor on the MFD indicates where the radar beam is centered to the crew within the F-15E cockpit. The cursor commands one of five functions, depending on which function is selected. There are five cursor functions: cue, target, update, mark, and map.

- The cueing function directs or commands a supporting imaging sensor, like Low Altitude Navigation and Targeting Infra-red for Night (LANTIRN) targeting forward-looking infrared (FLIR), to the current cursor location designated within the RBM mode.

- The targeting function cursor designates a target, allowing weapon attack steering symbology to display on the pilot's HUD.

- The updating function updates the mission navigator (MN) to ensure the display format symbols are positioned over the radar video as accurately as the system can provide.

- The marking function provides the pilot the capability to mark a specific point within the RBM mode for future reference.

- The mapping function directs the radar to prepare the system for commanding an HRM patch map.

The HRM mode incorporates many functions that operate like the RBM mode, with some additional functional capabilities specifically for high resolution mapping. The APG-70 radar is capable of producing high range and azimuth resolutions using synthetic aperture radar (SAR) mapping techniques. The SAR and the HRM are notably the same in the F-15E cockpit. The radar uses SAR principles to produce video output with a resolution capability many times that achievable with the RBM mode. Resolution is the minimum distance two objects can be separated and discerned as individual objects by the radar return output, rather than one large object. At a range of 20 NM the HRM mode provides resolution as good as 253 feet. For comparison purposes, the RBM mode at a range of 20 NM only provides a resolution of 5300 feet. In addition, the HRM mode provides better resolution at much greater ranges from an object when compared to the RBM mode.

An important factor affecting resolution is the size of the radar's antenna. The larger the antenna the better the resolution capability. The APG-70 radar "synthesizes" a much larger antenna by taking many snapshots of an area as the radar scan moves across the ground. Each snapshot is taken at a slightly different perspective (range and angle) due to the motion of the aircraft. The information is then gathered by the SAR and stored in computer memory. The image data is then processed by the central computer to produce high resolution patch maps of the desired area.

Two types of HRMs can be generated within the APG-70 radar system that is located in the F-15E aircraft, the HRM patch map and the HRM plan position indicator (PPI), which resembles the RBM PPI. A patch map provides a higher resolution map of a smaller area when contrasted with the PPI. The HRM patch map mode within the F-15E aircraft, can be commanded via the HRM PPI, RBM, or PVU modes.

A patch map is commanded by positioning the radar cursor over the desired area and by designating the area as the location to be mapped with the HOTAS. When the HRM mode is first selected, the radar presents the PPI format until a patch map is commanded within the F-15E aircraft. The radar then initiates the SAR processing and a patch map of the area is presented later in the process. The imagery presented on the patch map can then be visually searched for potential targets. If a potential target is identified, the pilot can then acquire additional patch maps of the area or cue EO/IR sensors to that point for further interpretation. The actual area, window size, that is mapped can be varied in increments of 0.67, 1.3, 3.3, 4.7, 10, 20, 40, and 80 NM, depending on the radar range that is currently selected. The range selections are the same as in the RBM mode (4.7, 10, 20, 40, 80, and 160 NM).

Within the F-15E's APG-70, there are two submodes that are available for an HRM map: stabilized, or progressive. When the stabilized option is selected, the radar continuously maps the selected point on the ground. When the progressive option is selected, the radar maintains a constant azimuth and range for mapping the desired area. Typically, patch maps are ground-stabilized (i.e., centered on a specific point of terrain). Once a map is obtained, the radar is capable of storing the two most current maps for recall and for use in the future.

2.2 APG-70 Pilot-Vehicle Interface Description

Design considerations were based on the IMPACT team's decision to first develop the real beam map (RBM) plan position indicator (PPI) and the high resolution map (HRM) patch map. Table 1 highlights the performance and purpose of the RBM and HRM modes of the F-15E's APG-70 radar system. The HRM patch map was developed to provide the basic SAR simulation capability in regards to mathematical modeling for the visual display. The RBM PPI mode was developed to provide the ability to have a representative simulation of an air-to-ground (A/G) radar that would provide the ability to display gross terrain features to assist pilots in navigation.

As reference data for future development of the APG-70 radar simulation, *Appendix A, Air-To-Ground Radar Pilot-Vehicle Interface* describes the basic A/G format, the common symbology among the radar modes, and the functionality of these modes as well. The

commonality between the RBM and the HRM modes is primarily within the PPI format. Immediately following Table 1, the RBM PPI and the HRM patch map modes are briefly described as these modes were the primary focus for the software simulation.

Table 1. APG-70 Primary Radar Modes

Mode	Performance		Purpose
Real Beam Map	**Resolution:** **Range:** **Azimuth:** **Maximum Range:**	127 FT 4.7 NM 2.5 deg 160 NM	• Gross Terrain Features for Navigation • Weather Detection • HRM Cueing
High Resolution Map	**Resolution:** to 20 NM **Resolution:** to 160 NM	8.5 FT (R/AZ) 127 FT (R/AZ)	• Wide Area Search • Position Updates • EQ Sensor Cueing • In-Weather Target Designation

2.2.1 Real Beam Map

The RBM mode is used to identify gross terrain features for navigation, to detect weather, and to cue the radar to a specific point for high resolution mapping, as presented in a PPI format. This format is selectable from the main menu, as shown in Figure 1. For clarification purposes, the bezel switches are numbered in a counter-clockwise fashion beginning with the bezel switch on the left side of the display underneath the BIT circle. Bezel switch number six is used to control the current mode of the APG-70 radar. The label above bezel switch six, displays 'RBM' to indicate the present mode as the real beam map mode (see Figure 1). However, it should be noted that the modes of the radar are also controlled via the HOTAS. The RBM mode takes approximately one second per sweep and provides six ranges: 4.7, 10, 20, 40, 80, and 160 NM. Also, the RBM incorporates the positionable cursor, sequence point data, and a means to transition to a high resolution map (called "patch map") presentation via HOTAS.

2.2.2 High Resolution Map

The HRM mode provides the capability to produce high range and azimuth resolutions using SAR mapping techniques. It provides the capability to produce both the PPI format and a patch map format (see Figure 2). The HRM PPI format is similar to the RBM PPI format, except it does not provide a real-time radar picture due to the delay in Doppler processing. Neither of the HRM formats provide radar returns for a $\pm$ 8 degree area, called the "blind

zone," around the aircraft ground track due to the Doppler notch. The HRM PPI radar image updates takes up to eighteen seconds per sweep versus one second in the RBM PPI. The highest angular resolution that can be obtained through Doppler processing occurs when the combination of Doppler frequency and angular rate of change is optimized. This occurs when the object is 90 degrees relative bearing from the aircraft, which leads to another restriction of the APG-70 radar system—its inability to reach the ideal situation of 90 degree azimuth due to its 60 degree azimuth limitation. HRM PPI ranges are the same as in the RBM PPI: 4.7, 10, 20, 40, 80, and 160 NM.

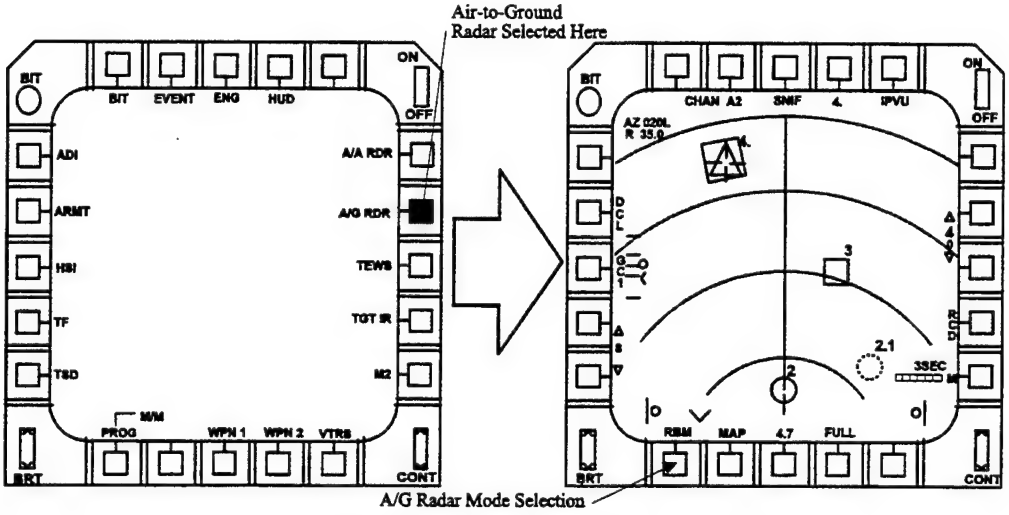


Figure 1. RBM Format Selected From Main Menu

Patch maps can be commanded from the RBM, the HRM PPI, or another patch map any time the cursor function is in "MAP." The process for making a patch map is to select the MAP cursor function at bezel switch 7, position the cursor over the area to be mapped, select the desired display window size using the auto acquisition switch on the throttle, ensure that the mapped area is not within the 8 degree blind zone and then press and release the throttle mounted Target Designation Controller (TDC). The display window size selected is the area that is designated to be the area of interest for the high resolution patch map. The parameters for the required minimum and maximum ranges, which are based on display window sizes, are listed in Table 2.

Table 2. HRM Patch Map Parameters on Display Window Size

Display Windows (NM)	Min/Max Mapping Range (NM)
0.67	4.7 / 20
1.3	4.7 / 40
3.3	4.7 / 50
4.7	4.7 / 80
10	10 / 160
20	20 / 160
40	40 / 160
80	80 / 160

As mentioned previously, a more detailed description of the modes of operation, including the HRM PPI and the HRM Patch Map, for the APG-70 radar are included within *Appendix A, Air-To-Ground Radar Pilot-Vehicle Interface.*

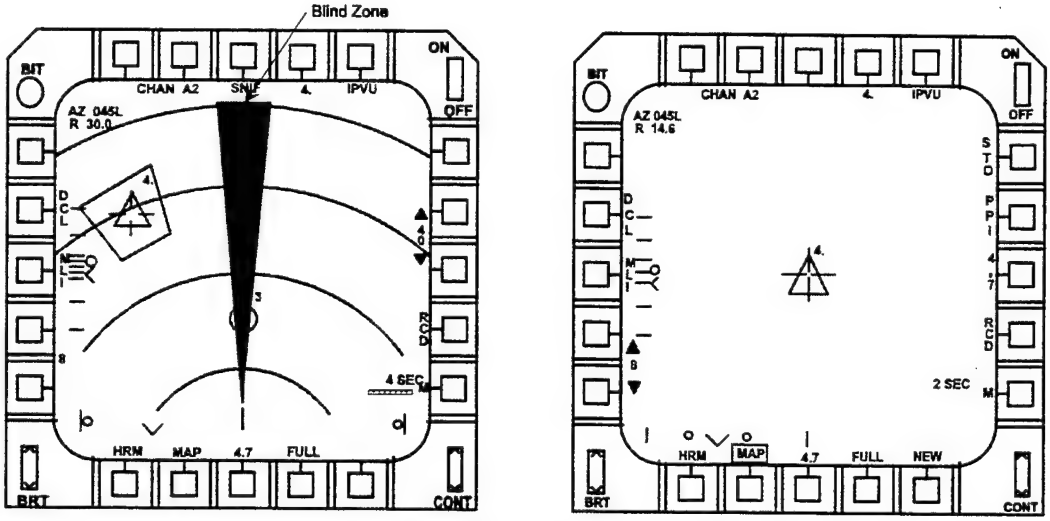


Figure 2. HRM PPI and HRM Patch Map Formats

3. SOFTWARE DESIGN CONSIDERATIONS

Design considerations were based on the IMPACT team's decision to first develop the real beam map (RBM) plan position indicator (PPI) and the high resolution map (HRM) patch map modes of the APG-70 radar. The following information provides an overview of the key technologies that were researched for the software implementation of the APG-70 radar simulation and the reasons why these technologies were not selected for development.

3.1 Software Libraries Reviewed

Several math libraries were downloaded and benchmarked to determine the fastest math library that would suit the needs for the APG-70 radar simulation of the real beam map (RBM) and synthetic aperture radar (SAR) modes. The forethought behind the strategy was to obtain a strong foundation of optimized math routines primarily for the vector math necessary for the graphical output produced by the APG-70 radar simulation. Two sets of packages were reviewed: one set included the public domain C++ packages and the other set included off-the-shelf products, namely the Iris Inventor and the Iris Performer Libraries. The public domain C++ libraries that were downloaded from the Internet and researched for the radar simulation were the Object-oriented Abstract Type Hierarchy (OATH), Library of Efficient Data types and Algorithms (LEDA), C++ Object Oriented Library (COOL), and the National Institute of Health Class Library (NIHCL). The public domain C++ packages, OATH and COOL appeared useful, but complex. Most of these math libraries would have taken a significant effort to integrate into the APG-70 radar due to the learning curve involved.

Although several libraries were generally reviewed, a strict comparison was performed between Iris Inventor and Iris Performer. Both of these libraries use some portion of the Iris Graphics Library (GL) as a foundation that provides an additional benefit, especially in comparison to the public domain libraries. Both products provide the similar functionality of culling the terrain database files, which means that ray intersections are performed against the polygons stored within the terrain database files so that a visual scene is generated from a

specific viewpoint. In other words, a scene is generated taking into consideration a specific three-dimensional (3D) viewpoint, ignoring information from the terrain database files which is not needed to render the scene (Hartman, J. and Creek, P. 1994). Iris Inventor has C++ interfaces to all its classes, however, the current version of Iris Performer does not. Iris Performer has most of its vector math routines implemented both as function calls and as macros, whereas Iris Inventor only has the member function calls. Iris Inventor benchmarked slower than Iris Performer.

For documentation purposes, the Open Graphics Library (GL) was partially considered instead of Iris GL to provide the drawing functionality needed by the radar simulation. However, Open GL was not heavily pursued at the beginning of the project for three reasons. First, Open GL was suspected to be slower than Iris GL. Second, Open GL was not yet promoted across the graphics industry when the project was initiated. And third, a learning curve was involved through the implementation path of Open GL. Therefore, the APG-70 radar simulation uses the standard Iris graphics library functions, Iris GL, to draw the radar output display images. In addition, the Iris GL functions are used to display the overlaying symbology for the RBM mode of the APG-70 radar simulation.

3.2 Texture Memory Benchmarked for RBM

Texture memory was considered for the implementation of the RBM PPI mode. Several experimental benchmarks of texture memory were completed to determine whether the technology would be fast enough for the RBM PPI of the APG-70 radar simulation.

The first experiments consisted of swapping images in and out of texture memory. These initial benchmarks tested images of the following byte sizes: 512x512x1, 480x480x1, 256x256x1, and 128x128x1. On the best machine available, the Onyx, each pass of 512x512x1 measured 0.20 seconds. Slightly better than linear speedup was obtained by reducing the size of the images, as indicated with the previously mentioned image sizes. In fact, the 128x128x1 byte sized images could be replaced at 104 times per second.

The second iteration of texture memory benchmarking involved the additional task of rendering the texture images onto polygons in a two dimensional (2D) double-buffered mode. The benchmark measured the time for mapping the image into texture memory and for rendering the polygons with the texture. The initial measurement involved the filling of one rendered polygon, a square window, with the largest image (512x512x1). Subsequent tests involved reducing the size of the image and replacing a varied number of the texture images in memory and also increasing the number of polygons to fill the same 2D area. The goal time was 0.067 seconds per pass with 0.100 seconds as the worse case. Unfortunately, the desired update rate was not achieved, disallowing the use of texture memory for the RBM PPI mode.

3.3 Wavelets Considered for RBM

One of the benefits of attending the SIGGRAPH '95 Conference was the identification of the wavelet technique. Wavelet technology is an extremely effective method of storing data currently used in applications that have to deal with large databases such as game software and film production. Since the RBM PPI mode of the radar simulation supports ranges from 4.7 to 160 nautical miles (NM), significant amounts of memory are needed to store the terrain data at a consistent resolution regardless of the currently selected range. Wavelets store information for the terrain database in an optimized fashion. With wavelets, the terrain data nearest the ownship is at a high level of detail, whereas the terrain data that is not as close to the ownship is not stored at a high level of detail. Therefore, wavelet technology optimizes the amount of memory used by the storage of the terrain database. The wavelet technique would have optimized the method in which the data is stored for the RBM mode of the APG-70 radar simulation, however, due to time constraints this technique was not implemented.

3.4 VSAR Reviewed

As a result of attending the 17th Interservice/Industry Training Systems and Education Conference (I/ITSEC) in 1995, information regarding the current state-of-the-art simulation environments and distributed interactive simulation (DIS) was obtained. Specific to the APG-70 radar simulation project was information about the virtual synthetic aperture radar

(VSAR) technique, which was presented in a proceedings paper entitled, *An Algorithm For Transforming Planned View Visual Imagery Into Synthetic Aperture Radar* (The American Defense Preparedness Association and The National Security Industrial Association, 1995). The algorithm was reviewed and also tested for its applicability to the APG-70 radar simulation.

The VSAR depends on two graphical processing passes of the terrain database: one pass is used to draw the visual image from an orthogonal viewpoint onto a screen, which stores the relative altitude within the Z buffer. The second pass reads the entire rasterized image back into memory and uses the Z buffer information to calculate the various shadows and SAR imagery. With the drawing of the visual image from an orthogonal viewpoint, the VSAR implementation must have a separate output window for the initial pass of the terrain database. In the testing of VSAR, Iris Performer was used to visually cull the orthogonal view of a flight file. This single need for an additional output window by the VSAR was the primary downfall of the VSAR and the reason for not using it within the final implementation of the APG-70 radar simulation of the HRM patch map mode.

3.5 Concluding Results of the Software Design Considerations

The texture memory benchmarking indicated that texture memory could not be used for the RBM PPI mode because of insufficient speed. Instead of using the wavelet technique or another investigated ray tracing technique, the final implementation of the APG-70 radar RBM mode obtains a radar sweep from the terrain database, processes the sweep of data located on the RBM radar model, and draws the simulated radar return into the display memory queues. The VSAR method was also not implemented for the HRM patch map mode because it required an additional output window which would limit the flexibility of the radar simulation and further complicate the integration. The specifics of the implemented software design, architecture, and implementation for both the RBM PPI and the HRM patch map modes are detailed within Sections 4, 5, and 6.

4. SOFTWARE DESIGN APPROACH

The choice to use object-oriented design in the software implementation of the radar simulation was necessary as this design technique more readily supports both changing requirements and the addition of new requirements. This object-oriented design technique stands as a major contrast to 'hard coding' techniques of the past. For speed, the Iris Graphics and Iris Performer libraries were chosen as the backbone for the vector mathematics and the graphical drawing of the radar output projections. Two complete iterations of configuration files were designed and implemented and are also detailed in this section. In addition, database utilities that were developed under the APG-70 radar project to primarily filter unnecessary data from the Defense Mapping Agency (DMA) terrain elevation and feature analysis database files are briefly mentioned in this section. Finally, the underlying mathematics for the simulated radar scan are discussed for the RBM PPI and HRM patch map modes of the APG-70 radar simulation.

4.1 Object-Oriented Design

Although object-oriented design was not a requirement of the software implementation of the APG-70 radar simulation, the design technique was chosen because of the flexibility and reusability offered through the object-oriented technology. The use of the C++ language was a requirement for the radar simulation model. Since C++ is an object-oriented language, using the object-oriented design approach was a natural decision to make. More benefits of utilizing object-oriented design include data abstraction, data encapsulation, inheritance, polymorphism, dynamic binding, aggregation, association relationships, object composition, delegation and parameterized types. In general, the object-oriented approach provides for source code reuse and effectively handles new requirements and modification of existing requirements (Booch, G 1994).

4.2 Iris Graphics and Iris Performer Libraries

Several math libraries were downloaded and benchmarked to determine the fastest math library that would suit the needs for the APG-70 radar simulation of the real beam map (RBM) and synthetic aperture radar (SAR) modes. A comparison among Iris Inventor and

Iris Performer was closely performed. Both of these libraries use some portion of the native Iris Graphics Library as a foundation providing an additional benefit, in comparison to the public domain libraries (see *Section 3, Software Design Considerations*).

As a result of the comparison, Iris Performer's math library was selected due to its speed, cost, and the fact that the product was already owned by CSIL. The APG-70 radar simulation uses the Iris Performer math library and vector data types as the math backbone of the RBM plan position indicator (PPI) and high resolution map (HRM) patch map mode simulations. The terrain databases are stored in memory using the Iris Performer data structures. However, the radar simulation does not use Performer to cull the terrain databases; this functionality is accomplished by functions provided within the simulation of the RBM and the HRM patch map modes of the APG-70 radar.

Although several libraries were researched for their vector math capabilities, Iris Graphics and Iris Performer, native libraries of the Silicon Graphics hardware platform were selected primarily for their optimized speed for drawing and culling. The two main libraries within Iris Performer are libpf and libpr. Libpr is a low-level library that provides high speed rendering functions (pfGeoSets), efficient graphics state control (pfGeoStates), and other application-neutral functions. Libpf is a real-time visual simulation environment that extends libpr to create a high-performance, multi-processing database rendering system taking advantage of the IRIS symmetric multiprocessing central processing unit (CPU) hardware (Fischler, S., et al. 1994).

For documentation purposes, Open Graphics Library (GL) was partially considered instead of Iris GL to provide the drawing functionality needed by the radar simulation. However, Open GL was not heavily pursued at the beginning of the project for several reasons as mentioned within *Section 3, Software Design Considerations*. Therefore, the APG-70 radar simulation uses the standard Iris graphics library functions, Iris GL, to draw the radar output display images. In addition, the Iris GL functions are used to display the overlaying symbology for the RBM mode of the APG-70 radar simulation.

4.3 Input Configuration Files

The truest form of the object-oriented philosophy assumes that all objects primarily communicate via message passing, such as in the Smalltalk language. An initial decision was made to follow the object-oriented philosophy and to provide this message passing capability as part of the objects within the radar simulation. As input into the radar system, configuration files would provide the messages to the radar system to establish which classes needed to be instantiated into objects for the initialization of the radar system. These configuration files would provide as much parameterization of the radar system as possible so that any parameter of the radar system could be varied by only changing a configuration file instead of source code.

Although this approach provides an extremely flexible design, several other software building blocks are required to handle this design. One significant building block is the design of a syntax to be handled by the configuration files of the radar system. An additional piece of software involves the ability to examine the configuration files for tokens and then to determine the meaning of the tokens found within the input configuration files. In addition, each class that can be instantiated via the configuration file must also have appropriate constructors or message receiving code that can be executed via the parser portion of the radar system. However, in light of the extra software building blocks required for this design approach, the software development team determined that input configuration files would be the communication method to the APG-70 radar simulation to provide the run-time flexibility needed by the radar system.

Although additional software had to be designed to support the input configuration files, the benefits of this design approach outweighed the initial cost estimates in terms of software development time. When initially considering the amount of software development time needed for an implementation to support the input configuration files, only a single implementation was considered. The initial implementation considered for the input configuration files was the message-based syntax and respective support code. In fact, after the second iteration of the GUF-based input configuration files, it is questionable as to whether the time spent developing the second implementation of the configuration files

outweighed the benefits. However, the second iteration of the input configuration files implementation is definitely an improvement over the first iteration. The details of the two implementations of the input configuration file syntax, namely message-based and GUF-based, are described in the following paragraphs.

4.3.1 Message-Based Configuration Files

The first version of the radar system input configuration file syntax, loosely based on the Smalltalk programming language, defined the radar system as a series of messages. The message passing between objects coincides directly with the truly object-oriented design philosophy. The first input configuration file syntax, namely, the message-based syntax, primarily used keywords. As indicated in the following syntax diagram, an optional instance name of the object is followed by ':=,' a keyword, and the optional argument list facilitating the construction of the object.

Message-Based Configuration File Syntax Diagram

```
[ optionalName := ]  keyword:  { optionalArgumentList }
```

For clarification purposes, a simple radar configuration file example follows using the message-based configuration file syntax includes the following significant keywords:

•Radar	•latitude	•longitude	•altitude	•RealBeamMode
•Display	•width	•height	•mode	•MgDmaDatabase
•Antenna	•elevationLimits	•maxRates	•beamWidth	•samplesPerSweep
•modes	•colorIndex	•scanWidths	•ranges	•sweepsPerDegree
•scanBars	•heading			

The following example defines a radar system, myRadarSystem, with a display window size of 480 by 480 pixels, provides the RBM mode, a physical antenna model, and a database. The RBM mode is defined with three sector scan widths and six radar ranges. The width and height display attributes for the RBM mode define where the image should be drawn on the display window via the Iris GL function, ortho2().

Example of Message-Based Configuration File

```
myRadarSystem := Radar: {
      latitude:          N37@00.0
      longitude:         W122@00.0
      altitude:          4000.0
      heading:           0.0
      rate:              20
      // Physical Display
      Display: {
            width:       480
            height:      480
            mode: "doublebuffer"
            colorIndex: 512
      #include "colors1.cf"
      }
      // Physical antenna model
      Antenna: {
      elevationLimits: { -60, 60 }// lower, upper
      maxRates:          { 60.0, 60.0 }// az, el
      beamWidth:         { 2.5, 2.5 }// horizonal, vertical
      }
      // Database
      MgDmaDatabase: {
      latitude:   N37@00.0
      longitude:  W122@00.0
      }
      modes: {
            modeOne := RealBeamMode: {
                  scanWidths: {100, 50, 25}
                  myRangeList := ranges: {4.7, 10, 20, 40, 80, 160}
                  scanBars: { 1, 2, 3, 4 }
      // define some constraints for this mode
      samplesPerSweep: 128
      sweepsPerDegree: 2
                  Display: {
                        width:       {-0,6, 0.6}
                        height:      {-0.1, 1.1}
                  }
            }
      }
}
```

The advantages of this message-based configuration file syntax are its flexibility, simplicity, and the fact that it is a context-free grammar. As this file is preprocessed via the C++ preprocessor, comments are permitted along with inclusion of other configuration files. In fact, the 'colors1.cf' file is a configuration file which defines the colors needed and can be replaced if a different color scheme was desired by including another file instead. The parser and the message class, RsMessage, are used to pass configuration information to the radar components within the simulation. The parser scans a configuration file and creates a list of messages that are sent to the radar simulation to instantiate each of the objects identified with

respective keywords. Each message contains an optional name, a keyword, and an argument value. In the example 'myRadarSystem := Radar { ... },' the parser constructs a RsMessage that contains the name 'myRadarSystem,' the keyword Radar, and a linked-list of other messages describing the radar system. In addition, the RsMessage class is used to pass data between radar components during the simulation execution. In this way, methods, also known as member functions, that are coded into a radar component to handle messages received from the parser can also be used to handle messages from other radar components during run-time.

Although a step in the right direction, there were several disadvantages to the message-based configuration file syntax and the respective implementation. The message-based configuration syntax was not based on any standard. In addition, the implementation of adding a new message keyword was cumbersome because each keyword had to be pre-defined in the RsMessage class. There was only limited data type checking because a reference pointer to an RsObject is passed in the RsMessage object. The message-based parser did not allow for backwards compatibility of the keywords.

As mentioned earlier, a software building block, needed to support the input configuration files, remains to support the specific syntax of the message-based configuration files. To provide background for the parser building block, the public domain software components used to facilitate the generation of the parser portion of the radar simulation, Bison++ and Flex++, are described briefly. The parser for both of the input configuration file syntaxes were developed similarly. Flex++ Version 2.3.8-7 and Bison++ Version 1.21-8 were the public domain software components used for both parser implementations. These software components were primarily chosen because of the C++ interface which they both provide. In addition, these software components are compatible with their counterparts YACC and LEX. Flex++ is the C++-based version of Flex. Flex is a fast lexical analyzer generator compatible with LEX. Bison++ is the C++-based version of Bison. Bison, compatible with YACC, is a general purpose parser generator that converts a grammar description for a context-free grammar into a C program to parse that grammar.

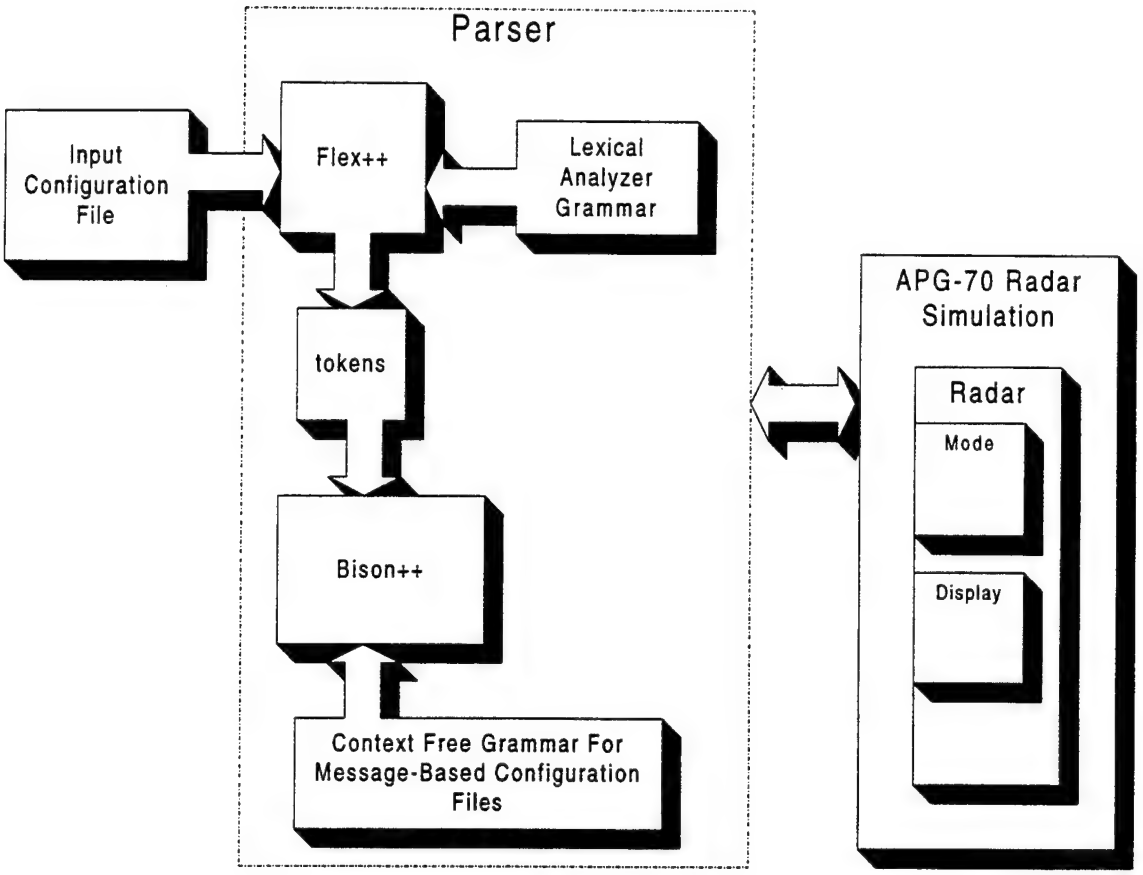


Figure 3. Data Flow Diagram for the Parser Portion of the APG-70 Radar Simulation

These two software components, Flex++ and Bison++, generally work very closely with each other (see Figure 3). Flex++ is a tool for recognizing lexical patterns in text such as the input configuration files. Flex++ reads a description of the lexical analyzer grammar and reads the input files. From the given grammar and the lexical patterns, also called tokens, within the input configuration files, Flex++ determines the tokens and provides these tokens to the Bison++ portion of the parser. Bison++ uses the output tokens from Flex++ and the context-free grammar rules provided to Bison++, to provide a parse() function that can be called to parse an input configuration file. The parse() function is called from the APG-70 radar simulation. Each object within the radar system that can be instantiated by the input configuration files must also provide a constructor with a signature to be used by the parser to create the radar objects at run-time using the input configuration files. These constructors instantiate the objects within the radar simulation, such as the display, the antenna, the

database, and the mode in the previous example. Once the radar simulation has initialized, all of the instantiated objects are then setup with their respective member functions, commonly named setup().

4.3.2 Grand Unified File Format-Based Configuration Files

The second and final version of the configuration syntax, similar to the Lisp programming language, is based on the Grand Unified File format (GUF) that was developed by Walt Disney Imagineering for use in movies, games, and simulator rides. The Silicon Graphics Performer development team is planning to use the GUF format in a later version of Iris Performer, as advertised at SIGGRAPH '95. In fact, GUF is a backwards compatible data exchange format that was designed with the following goals: ease of parsing, single pass construction, extensibility, and backwards compatibility (ACM SIGGRAPH 1995).

The following syntax diagram for basic GUF indicates that a pair of surrounding parenthesis encapsulates a form, 'formName,' and an argument list, 'argumentList,' which are space delimited.

Basic GUF Syntax Diagram

```
( formName argumentList )
```

An example follows to provide clarity of the basic GUF syntax. In the example, the name of the GUF form, 'formName,' is defined as 'rgb' and its respective argument list, 'argumentList,' include the values of the 'rgb' form's components. The form 'rgb' defines a color consistent of red, green, and blue real-valued number components.

Example of Basic GUF Syntax

```
( rgb :red 0.5 :green 0.6 :blue 0.7 )
```

In GUF, the parser reduces the list of zero or more arguments to a stream of slot pairs; each slot pair contains a slot name and value. Slot values can be numbers, Booleans, strings, identifiers, lists, or other forms. In the previous example, the slot values are real numbers. If a slot name is not provided, the parser assigns a name based on the position of the argument in the list. The second line of the following example provides the ordinal position of the slot

names, :1 and :2, which are defaulted because the slot name is not provided within the first line of the example.

Example Indicating Default Ordinal Position Values

```
( rgb 0.5 0.6 :blue 0.7 )
( rgb :1 0.5 :2 0.6 :blue 0.7 )
```

The parser uses the function formFunc() to construct a new instance of the form, in other words the formFunc() function instantiates the form identified with 'rgb.' All objects or classes which can be instantiated with the input configuration files, must have constructors available via the formFunc() member function. The parser then passes the stream of slot pairs to the new instance using the form's member function setSlot(). The forms that are used and defined by the radar system are as follows: rgb, rgba, hsv, hsva, lat-long, Mode, OffMode, StbyMode, RealBeamMode, System, Apg70, Display, Window, AntennaServo, RbmAntenna, Database, and DedDbase.

Lists are defined as simple GUF forms, guf, and can have one of two formats. The second format with the prefix ` is the format used predominantly within the APG-70 radar GUF-based configuration files as it is preferred due to its shorter length in comparison to the format which uses 'guf.'

Two Available GUF Form List Formats

```
      ( guf  1.0 2.0 3.0 )
or
      `( 1.0 2.0 3.0 )
```

A form can be defined to the parser using the def-form command. In the following example the new form 'rgb' is defined. The name for the form is 'rgb' and 'red,' 'green,' and 'blue' are the slot names that the newly defined form 'rgb' understands. The def-form command is not supported by the current implementation of the parser, however, an explanation is included here to illustrate the benefits of the GUF-based syntax.

Example of Defining a New Form

```
      ( def-form rgb  `red `green `blue )
```

Forms can be based on other forms providing backward compatibility. In the following example, the new form 'rgba' is defined. The new form 'rgba' contains all the slots of form 'rgb' plus the new slot 'alpha.'

Example of Backward Compatibility

```
( def-form rgba is= rgb 'alpha )
```

In the following example of a simple GUF-based radar configuration file, the radar system's form, Radar, has two slots, 'display' and 'modes.' The slot 'display' is set to form 'Window.' The slot 'modes' is set to a GUF list of only one form for the real beam mode (RBM).

Example of GUF-Based Configuration File

```
( Radar
      :display ( Window
            width:      480
            height:     480
            mode: "doublebuffer"
      )
      :modes '(
            ( RealBeamMode
                  :scanWidths { 100, 50, 25 }
                  :ranges { 4.7, 10, 20, 40, 80, 160 }
                  :Display ( RbmDisplay
                        width:      '( -0,6, 0.6 )
                        height:     '( -0.1, 1.1 )
                  )
            )
      )
)
```

An advantage of the GUF configuration syntax is that it is based on a future standard for the exchange of graphics information. In addition, it is much easier to expand due to its backward compatibility. As a result, the configuration file syntax can easily be expanded to support radar terrain and feature descriptions.

The RsMessage class, developed for the original message-based syntax, is no longer used to pass data between radar components. A set of base classes was developed to support the GUF parser that can be used to pass data between components using the setSlot() function.

Another advantage to the GUF-based configuration file method is the fact that data type checking is provided in contrast to the original message-based configuration file syntax. GUF support classes providing data type checking are provided within the Basic Library and include number, Boolean, and string classes.

If an 'rgba' form is passed to an older version of a GUF system that does not support it, the GUF parser will try the base form, 'rgb.' This ability of the GUF syntax can provide backward compatibility in the radar system. For example, a new version of the HRM mode simulation, form name 'HRMModeNew,' would be based on the original HRM simulation, 'HRMMode.' If a radar configuration file with 'HRMModeNew' is passed to a version of the radar system that does not support it, the radar system will continue to run, using the original HRM mode simulation.

4.4 MultiGen Files

The Defense Mapping Agency (DMA) provides Digital Terrain Elevation Data (DTED) files commonly used by applications such as the APG-70 radar simulation. MultiGen Incorporated provides a UNIX stand-alone utility, entitled 'readdma,' which reads the raw DMA DTED files and converts them to a post file format. The utility, 'readdma,' currently available within the CSIL facility, can be used to transfer raw DTED directly from tape or compact disk (CD) onto hard drives. Currently, the location of files which are already in the post file format within the IMPACT CSIL facility are located in the directory area entitled /disk2/mg/dted. One method of detecting that the files are already post-processed is the fact that the suffix of the filename is '.ded'. Each of the post-processed files consist of a file or cell, containing a grid of post values which indicate elevation at fixed intervals, providing a contour of the region of interest. Each cell represents a one degree squared area of the earth. These files are mentioned because the RBM plan position indicator mode of the radar simulation uses these files within the post file format to provide the wide area terrain databases needed. The names of the files currently used in the RBM radar simulation are as follows: N36W121.ded, N36W122.ded, N36W123.ded, N37W121.ded, N37W122.ded, N37W123.ded, N38W121.ded, N38W122.ded, and N38W123.ded.

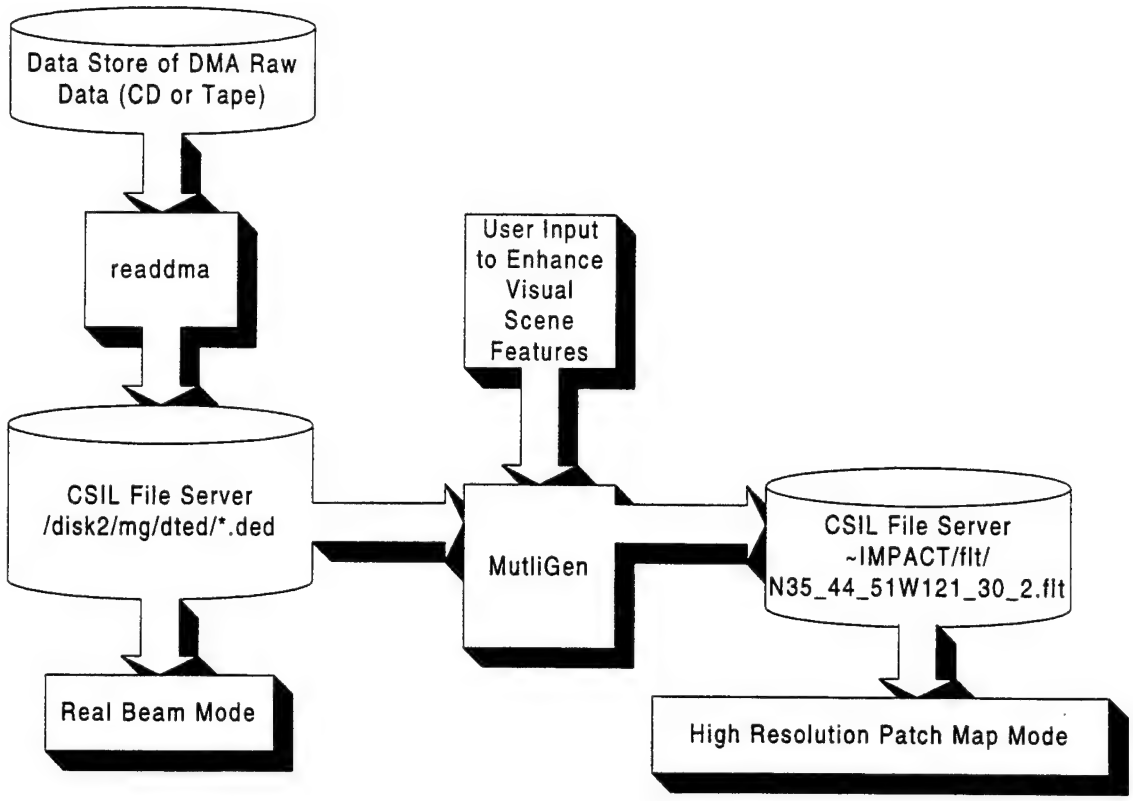


Figure 4. Data Flow Diagram for the DMA Data Files

The primary difference between the previously mentioned post-processed files and the flight files that are used to drive the visual scene output is that the flight files consist of polygons. MultiGen uses two different algorithms to take the post-processed formatted files and to convert them into the flight files. The Poly Mesh and Delaney algorithms implemented within MultiGen use the digital elevation post data and convert these data points into databases of polygons.

In addition, MultiGen is also a graphical interface design tool that allows the user to add 3D objects into the terrain databases to provide realism for the visual scene of the simulation (see Figure 4). In order to provide the high correlation between the visual scene and the terrain databases for the HRM mode, the APG-70 radar simulation system reads these databases from these MultiGen flight files that also drive the visual scene. For documentation purposes, the database file used for the IMPACT cockpit simulation is entitled N35_44_51W121_30_2.flt and the database reference coordinate is encoded into the name of the file, latitude position N35° 44' 51" and longitude position W121° 30' 2."

The database utilities which were developed under this project were developed to facilitate the filtering of the *.ded terrain database files used for the RBM PPI of the APG-70 radar simulation. One of the utilities outputs the data, ded_dump. The reduction of the *.ded terrain database files occurs with the ded_reduce utility and the snapshot of the *.ded terrain database files is seen via the ded_image utility.

4.5 Underlying Mathematical Representation of Simulated Radar Projections

The primary mission of the F-15E is all weather day and/or night interdiction. To accomplish this mission, the air-to-ground radar (APG-70) plays a significant role in accurate navigation to the target, precise target acquisition, and pinpoint weapons delivery. It is imperative that crew members understand basic radar principles and understand the radar capabilities and limitations. Effective use of the APG-70 radar depends on preflight planning in addition to a complete understanding of the radar scans displayed within the cockpit's MFD. Several inherent errors within the RBM PPI, such as beam width error and pulse length error, are simulated within the APG-70 radar simulation producing similar effects resulting with larger radar returns and therefore reducing the resolution capability.

In addition, within the cockpit, the MFD has a display resolution as well. The radar display consists of a 480 by 480 square area of pixels in order to display the radar scans. Dependent upon the currently selected range with the radar, the amount of data represented by a single pixel changes which affects the mechanical resolution of the radar scan. The following chart indicates the pixel size dependent on the selected radar range. Following the chart, within the next few paragraphs, the methods of simulating certain radar return effects are discussed briefly.

Range Scale(NM)	Pixel Size(feet)
4.7	59
10.0	127
20.0	253
40.0	507
80.0	1014
160.0	2028

4.5.1 Real Beam Map Plan Position Indicator Mode

The RBM PPI mode of the APG-70 radar simulation uses trigonometry in order to determine the simulated radar return from the DMA DTED files. As explained earlier, the DMA DTED files contain elevation posts at fixed intervals regarding the terrain database. A sweep of terrain elevation posts are measured with respect to the mean sea level (MSL). The position of the radar antenna and the radar beam source are assumed to be located in the nose of the aircraft at position A within Figure 5. The antenna elevation angle is measured from the horizon and is labeled as X within Figure 5. The aspect angle is the angle at which the center of the radar beam hits the tangent of the terrain and is labeled as Y within Figure 5. The F-15E's beam width is 2.5° also indicated within Figure 5 by the angle labeled Z. The elevation of the terrain at the point of incident (B) is measured as the vector CD. The elevation of the aircraft is measured as the vector AD. The tangent of the aspect angle Y is equal to the vector AC divided by the vector CB, also called the ground range. In addition, the vector AB, also called the slant range, can be determined through the use of the Pythagorean Theorem. To calculate the radar return intensity, the dot product of the aspect angle with the tan of the antenna elevation angle or look angle is calculated for each point within the radar sweep. The following equations reveal these relationships using the labels within Figure 5.

Equations

$AC=AD-CD$ $\qquad$ $\mathrm{Tan}(Y) = AC/BC$

$AC^2+BC^{2=} AB^2$ $\qquad$ Intensity of Radar Return = (AC/BC) DOT X

As the radar beam has a width of 2.5° degrees, half of the radar beam remains on either side of the center of the source. The width of the beam used to illuminate the target is based on the antenna size and is the most serious inherent limitation to the resolution. The error is always in azimuth and applies equally to both sides of the target making the radar return larger and producing a horizontal smearing effect. As the simulation scans a radar sweep of elevation posts, it keeps track of three radar sweeps, the current sweep, the previous sweep, and the sweep prior to the previous sweep. These three sweeps are used to perform a

weighted average of the radar return data to simulate the horizontal smearing effect caused due to the beam width error. The pulse length error is also inherent only to the RBM PPI mode of the APG-70 radar causing a vertical smearing effect within the radar return. To simulate this effect, within a single radar sweep, three of the elevation post calculations are maintained to perform another weighted average.

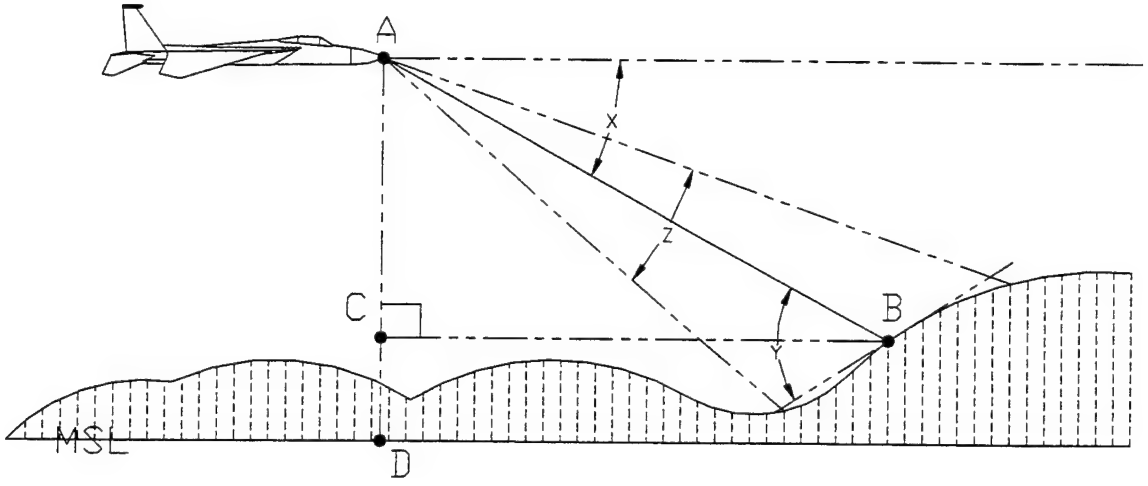


Figure 5. Antenna Elevation Angle(X), Aspect Angle(Y), and F-15E's Beam-Width(Z)

As many calculations are needed in order to determine the values of a complete radar return sweep, some effective filtering steps are implemented. If the tangent of the look angle is less than the minimum look angle value, Ymin, a zero is placed within the radar return sweep for that specific elevation post. If the tangent of the look angle is greater than the maximum look angle, Ymax, a zero is placed within the radar return sweep for that specific elevation post (see Figure 6). In this fashion, a minimum number of intensity calculations are performed directly speeding up the time required to determine the radar return sweep information.

In order to determine whether a specific elevation post is not seen because it occurs within a shadow, the tangent of the last most positive visible look angle is always maintained and denoted as value 'K.' When the current value of the tangent of the look angle is less than 'K,' it determines that a peak was just passed and therefore the current elevation post remains within a shadow (See Figure 7).

Uncontrollable factors which affect the radar display are the characteristics of topographic and cultural features such as, size, shape, and composition of objects within the terrain. Within the current simulation of the RBM PPI, this terrain culture information is defaulted.

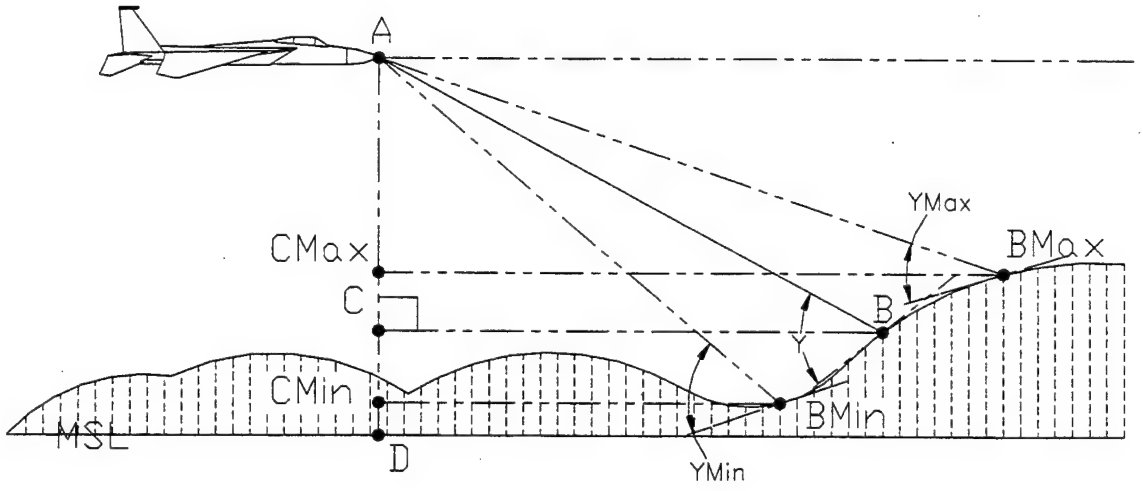


Figure 6. Maximum and Minimum Aspect Angles

4.5.2 High Resolution Patch Map Mode

One factor affecting resolution capability is the size of the antenna as noted within the previous paragraphs. The resolution capability of the HRM patch map mode is many times that of the resolution of the RBM mode due to the fact that the F-15E 'synthesizes' a 1400 foot long linear array of small antennas. Several snapshots are taken of the same area, ground stabilized, as the aircraft moves along its forward velocity vector flight path. Figure 8 shows an example of two snapshots, A1 and A2, being taken in order to produce an example patch map. The shadows and reflectivity within the patch map are obtained from the source of the radar with respect to the terrain information. However, the view of the patch map is an orthographic view from directly above the buildings almost at position A2 within Figure 8.

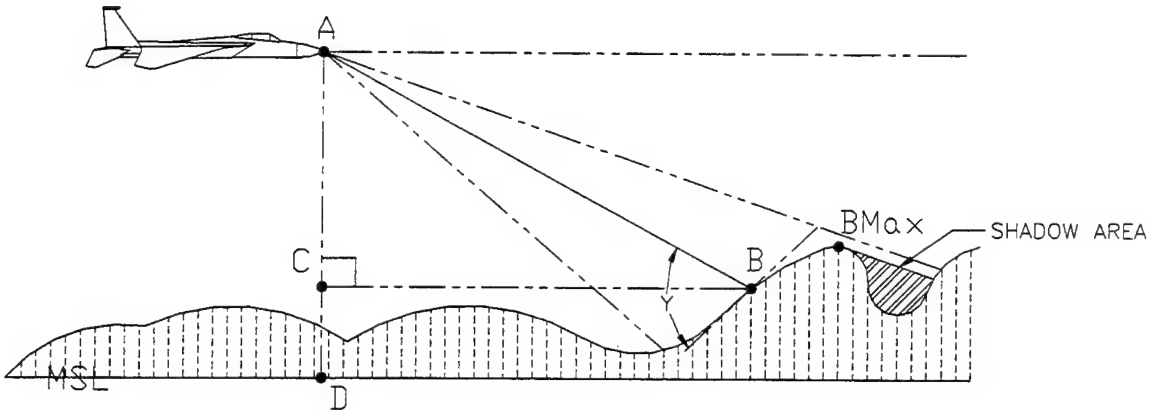


Figure 7. Radar Beam Shadows

In this mode, range is determined by measuring the time between transmission and reception of the radar pulse and azimuth is determined by doppler shift (change in frequency and rate of change in frequency). The doppler effect is a shift in the frequency of a wave reflected by an object in motion, the aircraft in this case. If objects within the terrain are moving, these objects will appear larger than they should be. Movements tend to make objects appear larger within the HRM patch map as all SAR techniques assume that objects are static. In order to simulate the HRM patch map output, polygonal MutiGen *.flt files are used as input and processed. Surface normals at the polygonal vertices are calculated and then interpolated from vertex to vertex using Phong shading. Once again, in order to calculate the radar intensity return information, the dot product of the look angle and the surface normals are calculated. A random number generator is used to add speckling effects to the HRM patch map output. Since no material reflectivity is stored within IMPACT's terrain database files, a manipulation of the color is used to simulate the material reflectivity of the output.

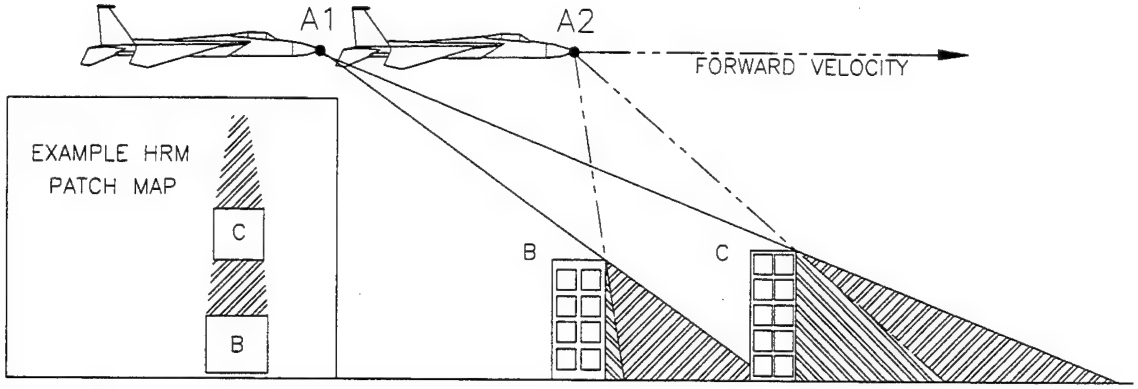


Figure 8. Example HRM Patch Map

5. SOFTWARE ARCHITECTURE INTRODUCTION

This section provides an initial data flow diagram inclusive of a complementary discussion of the software architecture underlying the APG-70 radar simulation. To directly orient a software person with the source code, explanations follow the data flow, such as an explanation of how the source code files are organized, how to recreate the executables, and how to run the APG-70 radar simulation.

5.1 Software Architecture Overview

The software architecture that provides the foundation for the APG-70 Radar model is best described with a data flow diagram, as shown in Figure 9. The arrows within the diagram indicate the direction of data flow between and among the various processes that support the viewing of the APG-70 radar format.

The APG-70 radar model developed under Delivery Order (DO) 0013 was primarily focused on the need for flexibility. Therefore, configuration files were used to support the flexibility within the radar model. The syntax within the radar model configuration files is based on the Grand Unified File (GUF) format by Walt Disney Imagineering. Details regarding the syntax of the GUF-based input configuration files are provided in *Section 4, Software Design Approach.* These input configuration files provide the instance information required for the initialization of the APG-70 radar model.

In order to integrate the APG-70 radar model into the CSIL IMPACT cockpit, two input configuration files are currently being used. One file, entitled apg70d.guf, primarily drives the displaying of the radar; the other file, entitled apg70.guf, drives the various modes of the radar, the selectable scanwidths, the radar ranges, the color setup, the display modes, the antenna model, the terrain database file types, and their respective locations.

The APG-70 radar parses the *.guf files and instantiates the objects dependent on the information within the configuration files. The APG-70 radar model uses Flex++ Version 2.3.8-7 and Bison++ Version 1.21-8 to generate the C++ files to support the parser

functionality. The parser selects the tokens from the configuration files, identifies the semantics from the GUF-based syntax, and proceeds to establish the instances of objects for the radar model. For more details on the parser, refer to *Section 4, Software Design Approach.*

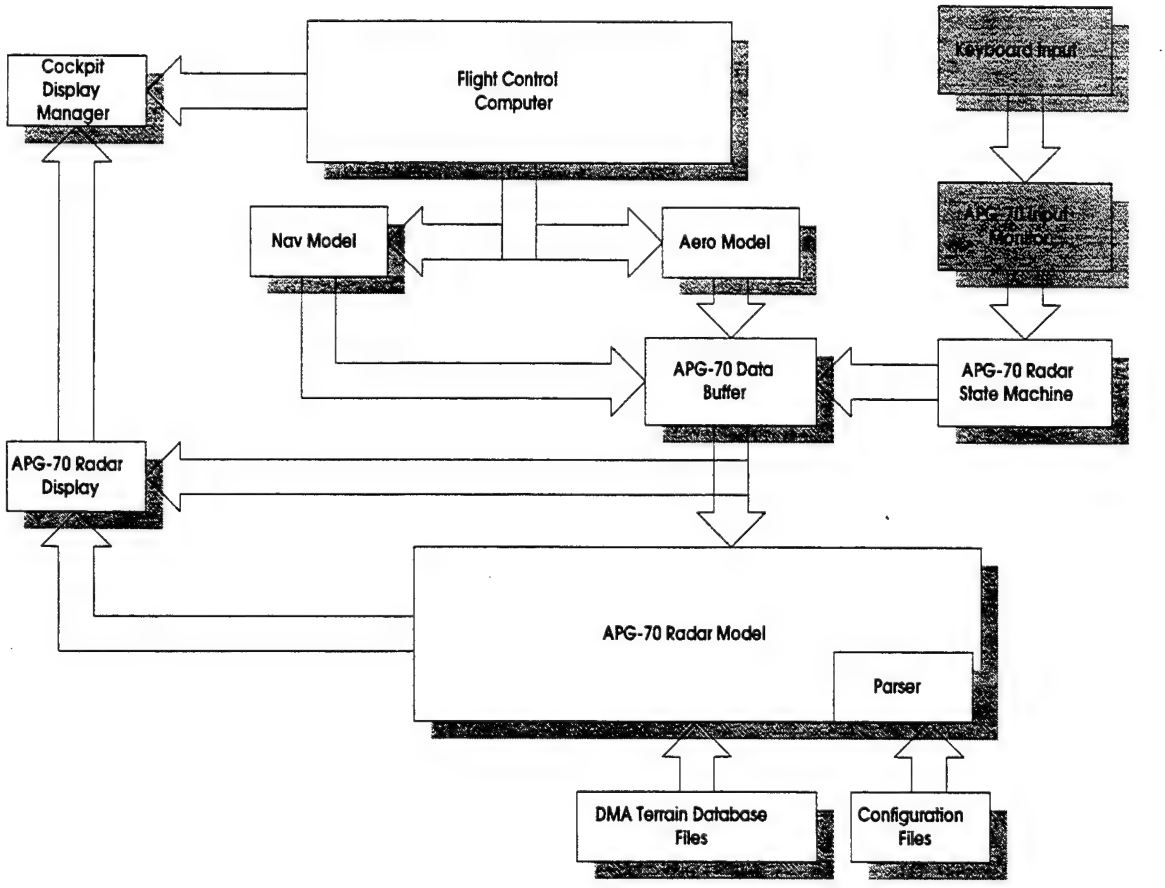


Figure 9. APG-70 Radar System Software Architecture

The integration of the APG-70 radar simulation system into the CSIL IMPACT cockpit is supported by the following processes: APG-70 radar model, APG-70 radar display, APG-70 data buffer, APG-70 state machine, and the APG-70 input monitor. Two of these processes are the foundation of the APG-70 radar simulation system, the APG-70 radar model and the APG-70 radar display. The radar model simulation process, APG-70 radar model, calculates the data to be displayed using the input configuration files, apg70.guf (with a display window) or apg70nd.guf (without a display window). The graphical drawing process, APG-70 radar display, renders the radar format using the input configuration file, apg70d.guf.

As shown in Figure 9, the output from the APG-70 radar model is sent to the APG-70 radar display. The APG-70 radar model outputs the calculated radar sweep data using the information it obtains from the APG-70 data buffer in addition to the DMA terrain database files. The DMA database files that feed the APG-70 radar model are either of the post processed format, *.ded files, for the RBM PPI mode or MultiGen polygonal flight files, *.flt, for the HRM patch map mode. The APG-70 data buffer gets information such as the aimpoints, steerpoints, and targets from the navigational (nav) model. The APG-70 data buffer also obtains snapshots of information from the aerodynamic (aero) model, such as the positioning of the aircraft in terms of latitude, longitude, and altitude so that the simulation of the radar display is realistically dynamic.

The radar sweep data is stored in shared memory allowing the APG-70 radar model process and the APG-70 radar display process to share the information. The update() routine within the APG-70 radar display for the drawing of the radar is called by the Cockpit Display Manager. The Cockpit Display Manager is also responsible for the drawing all other display formats within the cockpit. Additionally, the Cockpit Display Manager displays the bezel surrounding the APG-70 radar display and the other displays, as needed. The APG-70 radar display consists of the portion which draws the radar sweep data and also the portion which draws the overlaying graphical symbols. These graphical symbols are listed as follows: the bezel switch labels, the radar range arcs, the aimpoints, the steerpoints, the targets, the zero azimuth line, radar mode status, the selected cursor function, declutter status, gain control, display brightness, azimuth scan size, antenna azimuth and elevation, and radar range information. The bezel switch labels are drawn using the window set technique common to all formats currently integrated into the CSIL IMPACT cockpit.

Portions of the information displayed within the APG-70 radar display are indirectly determined by the APG-70 radar state machine because the state machine takes outputs from the APG-70 input monitor and the current state of the radar to determine the information needed within the APG-70 data buffer. As the APG-70 radar model and the APG-70 radar

display both use the information within the APG-70 data buffer to produce the radar sweep output and the radar display output, the APG-70 radar state machine stands as a vital portion of the APG-70 radar simulation system.

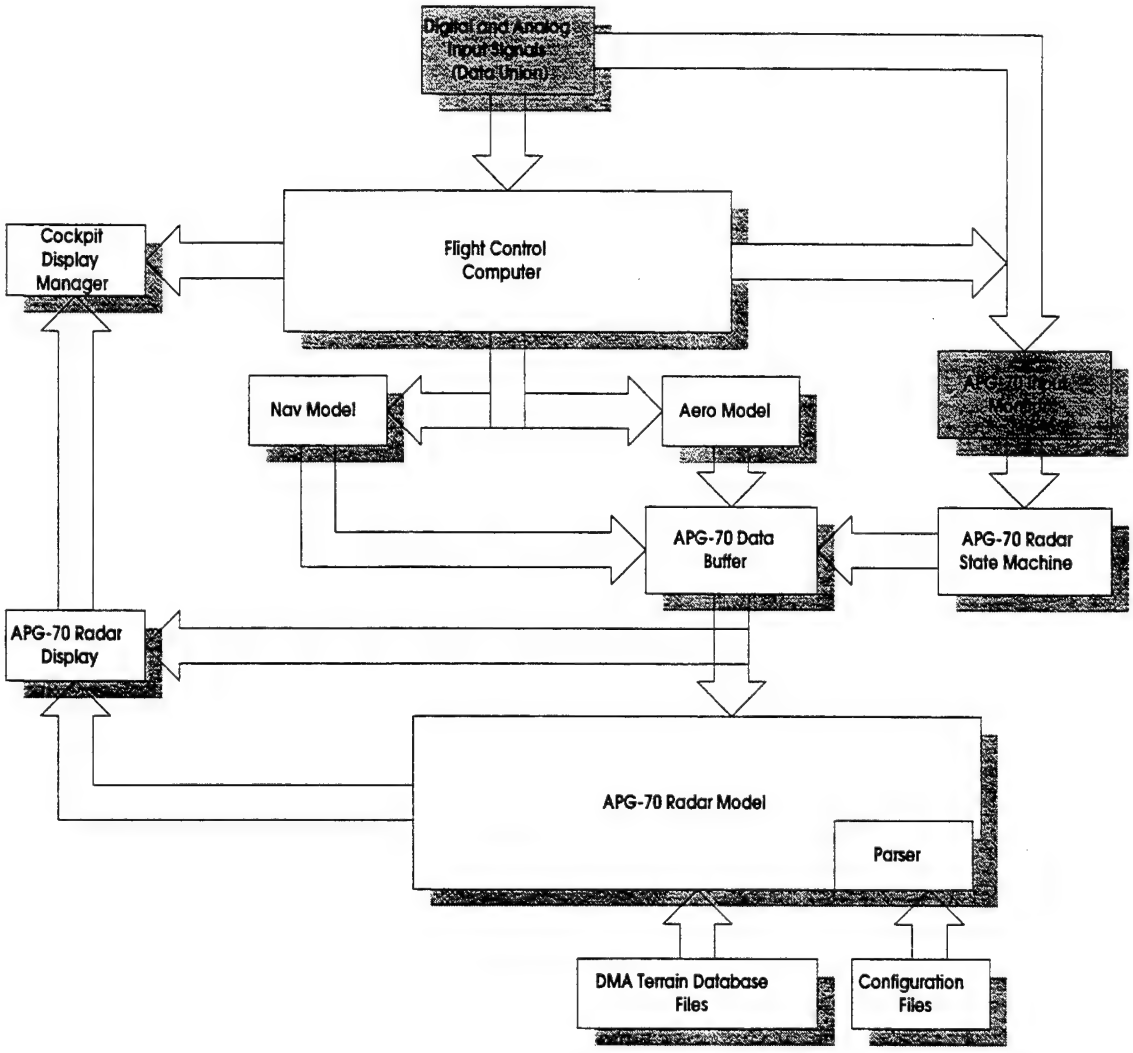


Figure 10. Complete Integration of the APG-70 Radar Simulation

To establish the correct functionality behind the APG-70 radar there exists an APG-70 radar state machine, which uses information provided by the APG-70 input monitor along with current state information of the APG-70 radar, stored within the wlAPG70DataBuffer object, to determine what to display. In the current integration state, keyboard inputs simulate the various bezel switches as needed for the APG-70 radar. These inputs are read with the APG-70 input monitor process which feeds into the APG-70 radar state machine. Upon final

integration of the APG-70 radar system, the Flight Control Computer process, fcc, will initiate all the processes needed to facilitate the simulation of the APG-70 radar model and all other processes needed to execute the entire cockpit simulation. Once the radar is completely integrated into the simulation, the hardware inputs which drive the radar simulation will be provided by the digital and analog signal output from the cockpit simulator via the data union rather than from a keyboard. (see Figure 10) In addition, a cursor will be driven by these signals to provide a method to command the HRM patch map from within the RBM PPI mode of the APG-70 radar simulation system.

5.2 Directory Structure

As this software simulates 'yet another radar simulation,' the acronym associated to the APG-70 radar system is defined as 'yars.' The 'yars' directory is revision controlled by the UNIX utility revision control system (RCS) at the /projects/IMPACT/yars location within the CSIL facility's server. Each directory listed in the following table (see Table 3) contains a set of files that are used to create a library of classes with a common theme or functional purpose. The $YARS identifies the /projects/IMPACT/yars directory, and $IMPACT identifies the /projects/IMPACT directory on the file server in the CSIL facility. The directories which are considered temporary or in a transitional state are moved into the $YARS/support directory. Upon final integration of the APG-70 radar simulation system, some of the information within this table may become outdated.

Table 3. APG-70 Radar Simulation Pertinent Directories

Directory Name	Purpose
$YARS/apg70	APG-70 radar system
$YARS/apg70/test	APG-70 radar test programs that instantiates an APG-70 radar simulation
$YARS/basic	Basic classes used to support GUF, parser, and generic applications
$YARS/cf	Final version of GUF-based configuration files
$YARS/data	Initial version of message-based configuration files
$YARS/database	Database access classes used to access various forms of databases using a standard application programming interface (API)
$YARS/database/ded	MultiGen*.ded file access classes
$YARS/database/perf	MultiGen/Performer *.flt file access classes
$YARS/database/ded/lowRes	Database utilities for RBM ded files
$YARS/database/ded/test	MultiGen DED file access classes test programs
$YARS/database/ded/tools	DED file utilities
$YARS/include	General header files
$YARS/lib	Libraries for all $YARS/*
$YARS/offMode	Off Mode class
$YARS/parser	Radar configuration file parser supporting GUF
$YARS/radar	Main radar abstract classes routines
$YARS/realBeamMode	Real Beam Mode specific classes
$YARS/stbyMode	Standby mode specific classes
$YARS/sarMode	SAR mode specific classes
$YARS/sarMode/test	SAR mode test programs for high resolution patch map mode
$IMPACT/mfd/src/wlAPG70DataBuffer	Data Buffer for APG-70
$IMPACT/mfd/src/wlAPG70DataBufferIMPACT	Concrete Data Buffer for APG-70
$IMPACT/mfd/src/wlAPG70OverlayGfx	Overlaying Graphical Symbols
$YARS/support/tests	Scramnet Access and test programs to set antenna elevation and to get waypoint and aerodynamic model information
$YARS/support/wlAPG70InputMonitor	Keyboard Input Monitor
$YARS/support/wlAPG70FormatVars	Initializes window set variables
$YARS/support/wlAPG70StateMachine	APG-70 State Machine class

5.3 Building Executables

To rebuild the executable programs for the APG-70 radar simulation, the proper revision of the source code is confirmed. The source code for the APG-70 radar simulation is stored with the UNIX utility RCS within CSIL's file server. All directories under the $YARS directory have their own Makefile, which is invoked by the UNIX command, make. At the $YARS directory, another Makefile loops through all of the subdirectories and their respective Makefiles. Therefore, to completely remake all of the $YARS libraries and executables, the directory is changed to the /lib and all library files must be removed. The directory is then changed back to the $YARS directory and then the UNIX make command is used by entering 'make' at the command line. The file named $YARS/include/makedefs defines the locations of the system, Iris Graphics, and Iris Performer libraries.

To make the IMPACT level source code, local copies of the files are needed and are checked-out with a lock via RCS. Once these files are modified and tested locally, the IMPACT level Makefiles are used to recreate the needed libraries. The additional source files for the radar input monitor, the radar state machine, and other support programs remain in their respective locations until further integration of the APG-70 radar simulation occurs. These processes supporting the radar simulation system are also rebuilt using the UNIX make command.

5.4 Executing the APG-70 Radar Simulation

Until the APG-70 radar simulation is completely integrated into the IMPACT cockpit, several processes run concurrently to view the APG-70 real beam map (RBM) plan position indicator (PPI) radar display. The flight control computer starts the navigational model and the aerodynamic model. These models place the navigational steering points and aircraft positional data in terms of latitude, longitude, altitude, and heading into memory. The navigational information is needed because the steering points overlay the radar return sweeps. To accept keyboard inputs, the APG-70 input monitor process runs by changing the directory to $YARS/support/wlAPG70InputMonitor and by running the program 'input_monitor' in the background. This program creates an Iris GL window, which waits for

keyboard input. The following list of keyboard input and respective commands simulates the bezel switches surrounding the APG-70 radar display. The following list is also available within the file named 'INFO' within the same directory as the wlAPG70InputMonitor code.

Key	Command
'1'	Nothing
'2'	Toggles Declutter
'3'	Gain Control Cycles: 3, 2, 1, 0
'4'	Increase Brightness 0 - 15
'5'	Decrease Brightness 0 - 15
'6'	Radar Mode Cycles: RBM, HRM, PVU, BCN
'7'	Cursor Function Cycles: MAP, UPDATE, TARGET, CUE
'8'	Does Nothing in RBM
'9'	Scanwidth Size Cycles: FULL, HALF, QTR
'a'	Does Nothing in RBM
'b'	Returns to Main Menu
'c'	Toggles Video Tape Recorder
'd'	Increase Range Cycles: 4.7, 10, 20, 40, 80, 160
'e'	Decrease Range Cycles: 4.7, 10, 20, 40, 80, 160
'f'	Does Nothing
'g'	Toggles IPVU Mode
'h'	Sequence Point Cycling
'i'	Toggles Sniff
'j'	Frequency Band Cycle 1 - 8
'k'	Frequency Channel Cycle A - E
'esc'	Exit APG-70 Input Monitor

A location within Scramnet memory is filled with a value that simulates the antenna elevation. To place a value within the antenna elevation memory location, the directory is changed to the $YARS/support/tests directory. The 'aelev' program runs at the command prompt with a value for the antenna elevation parameter. Typically, a value of -20 is the input parameter to the 'aelev' program. The APG-70 radar state machine runs in order to initialize the window set variables and to instantiate the needed classes. To run the APG-70 radar state machine process, the directory is changed to $YARS/support /wlAPG70StateMachine, and then run 'st_machine' in the foreground at the command prompt. The APG-70 radar state machine process runs continuously, until it is killed by the operating system. To initiate the running of the APG-70 radar model, the directory should be changed to $IMPACT/yars/apg70. The following command within this directory initializes

the APG-70 radar model 'apg70 -f apg70rbm &' and runs it as a background process with a display window. To run a display only view of the APG-70 radar simulation within the IMPACT cockpit, the IMPACT 'mfd' program is run with the APG-70 radar configuration file. Upon final integration of the APG-70 RBM PPI mode, the information within this paragraph may become outdated.

As the HRM patch map mode is not as integrated as the RBM PPI mode, the HRM patch map mode runs primarily in a stand-alone fashion. The directory is changed to the $IMPACT/yars2/sarMode/test and the apg70hrm program should be run in the background with the following command 'apg70hrm -f apg70hrm &.' This previous command brings up a single SAR view of the MultiGen polygonal terrain database file used by the visual scene of the IMPACT cockpit. Upon final integration of the APG-70 HRM mode, the information within this paragraph may become outdated.

6. SOFTWARE IMPLEMENTATION

The following sections describe the various libraries that comprise the APG-70 radar simulation source code. Class hierarchy diagrams are used in order to better illustrate the inheritance relationships among the classes.

6.1 Basic Library

The Basic library, denoted libBasic, primarily supports the ability to maintain and handle the GUF-based language used by the input configuration files (see Figure 11). The input configuration files provide maximum flexibility and reconfigurability of the radar simulation without the need to modify source code, re-compile source code, or re-link object files. The input configuration files for the radar system use the GUF-based language to instantiate the various components of the radar system upon initialization. In addition to establishing the objects that are needed for the radar simulation, the various parameters for the radar and its subsystems are also provided within the input configuration files. Therefore, the modification of an input configuration file can actually change the radar system parameters so that the radar selectable ranges and antenna scan widths differ. In fact, the radar simulation could more closely represent the parameters of a different air-to-ground radar system if the input configuration files were greatly modified. In fact, the APG-68 radar system which exists within the F-16 Fighting Falcon, has a RBM PPI mode that only varies slightly compared to the F-15E's RBM PPI with respect to the selectable ranges and the scan width coverages. The range scale selections of 10, 20, 40, and 80 nautical miles (NM) within the F-16 compare to the F-15E's 4.7, 10, 20, 40, 80, and 160 NMs range selections. The scan width coverage of the F-15E has selections for 100, 50, and 25 degrees in contrast to the scan width coverages of the F-16: 120, 60, and 20 degrees. The purpose of this example is to indicate that only with the change of an input configuration file, a wide variety of radar simulations can be executed due to the flexibility of the current software implementation.

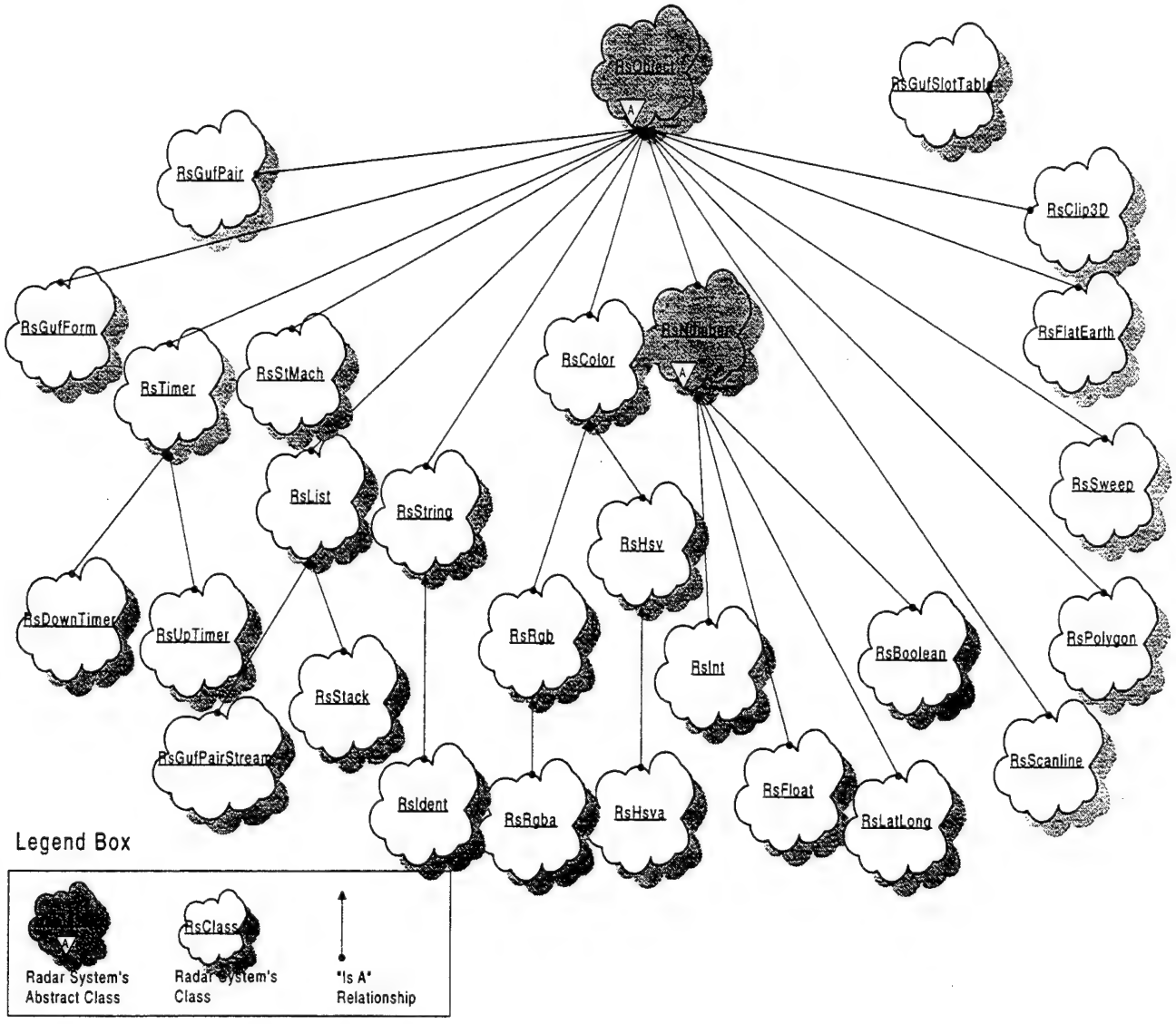


Figure 11. Class Hierarchy Diagram for Basic Library

Since each object within the configuration file has a specific set of related information, the parser for the radar system must be able to identify the various tokens within the configuration file. Once the tokens are identified by the lexical analyzer portion of the parser, the meaning of the token must be established. Tokens can be specific to the Grand Unified File (GUF) format syntax or they may be values of the various parameters supported by the radar system. Since the tokens may represent values, the various types of parameters must have classes for instantiation. A significant portion of the Basic library supports value-based types, such as RsString, RsNumber, RsColor, and their respective subclasses. Figure 11

shows the class hierarchy of the Basic Library using Booch's notation for object-oriented design (Booch, G. 1994). In addition to supporting the GUF-based syntax, these classes are also used within the subsystems of the radar system. The RsGufForm, RsGufPair and RsGufPairStream container classes support the GUF-based syntax used by the input configuration files.

The rest of the classes within the Basic library provide basic functionality that can be used outside of the radar system. These classes, namely RsTimer, RsList, RsStack, RsStMach, RsPolygon, RsFlatEarth, and RsClip3D, provide support for timers, state machines, stacks, doubly-linked lists, polygon storage, flat earth database modeling, and polygon clipping algorithms. In addition, the data store that is used to represent the radar output information is handled with the RsSweep class.

As mentioned within the legend box, the Rs prefix to the class names indicates that the class was implemented for the Radar System. In addition, almost all classes internal to the radar system are derived subclasses from the RsObject class. For example, within the Basic library, the RsGufSlotTable class is an example of a base class that does not derive from the RsObject class. Classes that were used to perform the integration of the radar system within the CSIL facility use the 'wlAPG70' prefix which stands for Wright Laboratory's APG-70 source code.

The initial version of the radar system used messages as the primary method to pass data from the configuration files to the radar component classes. Therefore, a RsMessage class was created to handle the inter-object communication that is similar to the Smalltalk language. The fact that all objects were derived subclasses of the RsObject class, made the passing of information much easier. This design choice of all radar system classes being subclassed from the RsObject class remains within the second version of the radar system code. The RsMessage object contained a pointer (or reference) to a RsObject, which was passed around the instantiated objects until the destination object took ownership of the RsMessage. Unfortunately, within the initial version of the software, type checking of the RsMessage data was not built into the implementation since a reference pointer was used. In

the current radar version of the software, the RsMessage class is only used by RsMemQueue to pass data messages via a common memory area. Based on the message type, the message can contain a number, a string, void pointer, RsSweep, or a link-list and type-checking is actually built into the implementation of passing data among objects. An RsMessage's type is determined when it is constructed or instantiated and cannot be changed after initialization.

To enhance the understanding of the Basic library, each individual class and its respective public interface is explained in the following paragraphs.

6.1.1 RsObject

The RsObject class stands as an abstract base class for the majority of the radar system (Rs-prefixed), classes. An abstract base class is not instantiable, however, it does provide common data and behavior for the derived sub-classes. The RsObject class provides type checking and support for the GUF forms. However, not all RsObject class derivations are GUF forms, but all RsObject class derivations can support the GUF forms. The instances of the RsObject class derivations that are GUF forms contain a slot table.

The public interface of the RsObject class follows:

```
RsObject(void);
virtual~RsObject();
int isGuf(void) const;
int isForm(void) const;
virtual RsType classType(void) const = 0;
virtual int isClassType(RsType type) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
const char* getSlotName(const int slotindex) const;
int getSlotIndex(const char* slotname) const;
void setSlot(const char* slotname,const RsObject* obj);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
RsObject* getSlot(const char* slotname) const;
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
void indent(ostream& sout, const int ident);
```

6.1.2 RsNumber

The RsNumber class stands as an abstract base class for numbers, such as floats, integers, Booleans, and Latitude-Longitude measurements. The RsNumber class derives from the RsObject class. The RsNumber class provides the common behavior for its derived classes

through the virtual member functions, such as getDouble(), getFloat(), getInt(), getBoolean(), classType(), and isClassType().

The public interface of the RsNumber class follows:

```
RsNumber(void);
virtual ~RsNumber();
virtual double getDouble(void) const = 0;
virtual float getFloat(void) const = 0;
virtual int getInt(void) const = 0;
virtual int getBoolean(void) const = 0;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.3 RsBoolean

The RsBoolean class stands as a class of Boolean numbers. An instance of the RsBoolean class maintains a value of TRUE (one) or FALSE (zero). The RsBoolean class derives from the RsNumber class. The RsBoolean class provides several operators as follows: assignment, equals, not equals, logical and, logical or, logical not, stream input, and stream output. The derived virtual classes from the RsNumber superclass are declared within the RsBoolean public interface as getDouble(), getFloat(), getInt(), getBoolean(), classType(), and isClassType() and are implemented within the RsBoolean's internals.

The public interface of the RsBoolean class follows:

```
RsBoolean(const int);
RsBoolean(const RsBoolean&);
RsBoolean(void);
virtual ~RsBoolean();
operator int(void)const;
virtual double getDouble(void) const;
virtual float getFloat(void) const;
virtual int getInt(void) const;
virtual int getBoolean(void) const;
RsBoolean& operator=(const RsBoolean&);
RsBoolean& operator=(const int);
friend int operator==(const RsBoolean&, const RsBoolean&);
friend int operator==(const int, const RsBoolean&);
friend int operator==(const RsBoolean&, const int);
friend int operator!=(const RsBoolean&, const RsBoolean&);
friend int operator!=(const int, const RsBoolean&);
friend int operator!=(const RsBoolean&, const int);
friend int operator&&(const RsBoolean&, const RsBoolean&);
friend int operator&&(const int, const RsBoolean&);
friend int operator&&(const RsBoolean&, const int);
friend int operator||(const RsBoolean&, const RsBoolean&);
friend int operator||(const int, const RsBoolean&);
friend int operator||(const RsBoolean&, const int);
```

```
friend int operator!(const RsBoolean&);
friend istream& operator>>(istream& stream, RsBoolean& number);
friend ostream& operator<<(ostream& stream, const RsBoolean& number);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.4 RsInt

The RsInt class stands as a class of integer numbers. An instance of the RsInt class maintains a whole number value within the range of -MAXINT to +MAXINT. The RsInt class derives from the RsNumber class. MAXINT is the maximum integer value provided by the implementation of the IRIX 5.2 operating system. The RsInt class provides several operators as follows: assignment, unary addition, binary addition, unary subtraction, binary subtraction, unary multiplication, binary multiplication, unary division, binary division, equals, not equals, less than or equal to, less than, greater than or equal to, greater than, stream input and stream output. The derived virtual classes from the RsNumber superclass are declared within the RsInt public interface as getDouble(), getFloat(), getInt(), getBoolean(), classType(), and isClassType() and are implemented within the RsInt's internals.

The public interface of the RsInt class follows:

```
RsInt(const int);
RsInt(const RsInt&);
RsInt(void);
virtual ~RsInt();
operator int(void) const;
virtual double getDouble(void) const;
virtual float getFloat(void) const;
virtual int getInt(void) const;
virtual int getBoolean(void) const;
RsInt& operator=(const RsInt&);
RsInt& operator=(const int);
void operator+=(const RsInt&);
void operator+=(const int);
friend int operator+(const RsInt&, const RsInt&);
friend int operator+(const int, const RsInt&);
friend int operator+(const RsInt&, const int);
void operator-=(const RsInt&);
void operator-=(const int);
friend int operator-(const RsInt&, const RsInt&);
friend int operator-(const int, const RsInt&);
friend int operator-(const RsInt&, const int);
void operator*=(const RsInt&);
void operator*=(const int);
friend int operator*(const RsInt&, const RsInt&);
friend int operator*(const int, const RsInt&);
```

```
friend int operator*(const RsInt&, const int);
void operator/=(const RsInt&);
void operator/=(const int);
friend int operator/(const RsInt&, const RsInt&);
friend int operator/(const int, const RsInt&);
friend int operator/(const RsInt&, const int);
friend int operator==(const RsInt&, const RsInt&);
friend int operator==(const int, const RsInt&);
friend int operator==(const RsInt&, const int);
friend int operator!=(const RsInt&, const RsInt&);
friend int operator!=(const int, const RsInt&);
friend int operator!=(const RsInt&, const int);
friend int operator<(const RsInt&, const RsInt&);
friend int operator<(const int, const RsInt&);
friend int operator<(const RsInt&, const int);
friend int operator<=(const RsInt&, const RsInt&);
friend int operator<=(const int, const RsInt&);
friend int operator<=(const RsInt&, const int);
friend int operator>(const RsInt&, const RsInt&);
friend int operator>(const int, const RsInt&);
friend int operator>(const RsInt&, const int);
friend int operator>=(const RsInt&, const RsInt&);
friend int operator>=(const int, const RsInt&);
friend int operator>=(const RsInt&, const int);
friend istream& operator>>(istream& stream, RsInt& number);
friend ostream& operator<<(ostream& stream, const RsInt& number);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.5 RsFloat

The RsFloat class stands as a class of floating point numbers. An instance of the RsFloat class maintains a value of a signed real number. The RsFloat class derives from the RsNumber class. The RsFloat class provides several operators as follows: assignment, unary addition, binary addition, unary subtraction, binary subtraction, unary multiplication, binary multiplication, unary division, binary division, equals, not equals, less than or equal to, less than, greater than or equal to, greater than, stream input and stream output. The derived virtual classes from the RsNumber superclass are declared within the RsFloat public interface as getDouble(), getFloat(), getInt(), getBoolean(), classType(), and isClassType() and are implemented within the RsFloat's internals.

The public interface of the RsFloat class follows:

```
RsFloat(const float);
RsFloat(const RsFloat&);
RsFloat(void);
virtual ~RsFloat();
operator float(void) const;
virtual double getDouble(void) const;
```

```
virtual float getFloat(void) const;
virtual int getInt(void) const;
virtual int getBoolean(void) const;
RsFloat& operator=(const RsFloat&);
RsFloat& operator=(const float);
void operator+=(const RsFloat&);
void operator+=(const float);
friend float operator+(const RsFloat&, const RsFloat&);
friend float operator+(const float, const RsFloat&);
friend float operator+(const RsFloat&, const float);
void operator-=(const RsFloat&);
void operator-=(const float);
friend float operator-(const RsFloat&, const RsFloat&);
friend float operator-(const float, const RsFloat&);
friend float operator-(const RsFloat&, const float);
void operator*=(const RsFloat&);
void operator*=(const float);
friend float operator*(const RsFloat&, const RsFloat&);
friend float operator*(const float, const RsFloat&);
friend float operator*(const RsFloat&, const float);
void operator/=(const RsFloat&);
void operator/=(const float);
friend float operator/(const RsFloat&, const RsFloat&);
friend float operator/(const float, const RsFloat&);
friend float operator/(const RsFloat&, const float);
friend int operator==(const RsFloat&, const RsFloat&);
friend int operator==(const float, const RsFloat&);
friend int operator==(const RsFloat&, const float);
friend int operator!=(const RsFloat&, const RsFloat&);
friend int operator!=(const float, const RsFloat&);
friend int operator!=(const RsFloat&, const float);
friend int operator<(const RsFloat&, const RsFloat&);
friend int operator<(const float, const RsFloat&);
friend int operator<(const RsFloat&, const float);
friend int operator<=(const RsFloat&, const RsFloat&);
friend int operator<=(const float, const RsFloat&);
friend int operator<=(const RsFloat&, const float);
friend int operator>(const RsFloat&, const RsFloat&);
friend int operator>(const float, const RsFloat&);
friend int operator>(const RsFloat&, const float);
friend int operator>=(const RsFloat&, const RsFloat&);
friend int operator>=(const float, const RsFloat&);
friend int operator>=(const RsFloat&, const float);
friend istream& operator>>(istream& stream, RsFloat& number);
friend ostream& operator<<(ostream& stream, const RsFloat& number);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.6 RsLatLong

The RsLatLong class stands as a class of numbers whose values represent a latitude or a longitude measurement. Each instance of the RsLatLong class contains a condensed numerical value, whose components consist of an enumerated direction, and values for the following measurements: degrees, minutes, and seconds. The RsLatLong class derives from the RsNumber class. The RsLatLong class contains a slot table that indicates it is a GUF

form. There are four components within the slot table for the RsLatLong class as indicated in Table 4. The derived virtual classes from the RsNumber superclass are declared within the RsFloat public interface as getDouble(), getFloat(), getInt(), getBoolean(), classType(), and isClassType() and are implemented within the RsLatLong's internals. An enumerated type, Dir, is used within this class to indicate the direction of an instantiated object. For the support of GUF, certain functions are provided within the public interface of the RsLatLong class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). To comply with the data encapsulation feature of object-oriented code, the following functions provide an interface to obtain the four components of an RsLatLong object: getDir(), getDeg(), getMin(), and getSec().

Table 4. RsLatLong's GUF Slot Table

Slot Index	Slot Name	Value Type
1	direction	enumerated direction
2	degrees	double
3	minutes	float
4	seconds	float

The public interface of the RsLatLong class follows:

```
enum Dir { none, north, south, east, west };
RsLatLong(const RsLatLong&);
RsLatLong(void);
virtual ~RsLatLong();
operator double(void) const;
virtual double getDouble(void) const;
virtual float getFloat(void) const;
virtual int getInt(void) const;
virtual int getBoolean(void) const;
Dir getDir(void) const;
int getDeg(void) const;
int getMin(void) const;
float getSec(void) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual void print(ostream& sout, const int indent = 0,
          const int slotsOnly = FALSE)
static const char* form;
static const RsType type;
```

6.1.7 RsString

The RsString class stands as a class of single byte character strings. The RsString class provides the ability to manipulate and obtain information from byte character strings. The RsString class provides the unique ability to convert between numbers and byte characters strings when applicable. Several member functions are provided via the public interface to facilitate the converting between numbers and byte character strings, such as len(), isEmpty(), empty(), getString(), isInteger(), isNumber(), getInteger(), and getNumber(). The RsString class provides several operators as follows: assignment, unary concatenation, binary concatenation, equals, not equals, less than or equal to, less than, greater than or equal to, greater than, stream input and stream output. The RsString class also handles its own memory management through the new and delete operators. Several constructors exist for the RsString class to provide for maximum flexibility. The RsString class derives from the RsObject class.

The public interface of the RsString class follows:

```
RsString(const RsString& string);
RsString(const char* string);
RsString(const char* s1, const char* s2);
RsString(void);
virtual ~RsString(void);
operator const char*(void) const;
int len(void) const;
int isEmpty(void) const;
void empty(void);
char* getString(void) const;
int isInteger(void) const;
int isNumber(void) const;
int getInteger(void) const;
double getNumber(void) const;
RsString& operator=(const char*);
RsString& operator=(const RsString&);
void operator+=(const char*);
friend RsString operator+(const RsString&, const RsString&);
friend RsString operator+(const char*, const RsString&);
friend RsString operator+(const RsString&, const char*);
friend int operator==(const RsString&, const RsString&);
friend int operator==(const char*, const RsString&);
friend int operator==(const RsString&, const char*);
friend int operator!=(const RsString&, const RsString&);
friend int operator!=(const char*, const RsString&);
friend int operator!=(const RsString&, const char*);
friend int operator<(const RsString&, const RsString&);
friend int operator<(const char*, const RsString&);
friend int operator<(const RsString&, const char*);
friend int operator<=(const RsString&, const RsString&);
friend int operator<=(const char*, const RsString&);
```

```
friend int operator<=(const RsString&, const char*);
friend int operator>(const RsString&, const RsString&);
friend int operator>(const char*, const RsString&);
friend int operator>(const RsString&, const char*);
friend int operator>=(const RsString&, const RsString&);
friend int operator>=(const char*, const RsString&);
friend int operator>=(const RsString&, const char*);
friend istream& operator>>(istream& stream, RsString& string);
friend ostream& operator<<(ostream& stream, const RsString& string);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.8 RsIdent

The RsIdent class stands as a class of identifiers. The identifiers class operates like the RsString class. Each instance of the RsIdent class behaves like the RsString class; however, it is likely that each instance of the RsIdent class will contain an identifier or a name that may not convert into a number. The RsIdent class derives from the RsString class.

The public interface of the RsIdent class follows:

```
RsIdent(const RsIdent& string);
RsIdent(const char* string);
virtual ~RsIdent(void);
RsIdent& operator=(const RsIdent&);
friend ostream& operator<<(ostream& stream, const RsIdent& string);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.9 RsStMach

The RsStMach class stands as a class of general purpose state machines. Instances of the RsStMach class can be used to control sequencing of events, modes, or submodes. Each instance of the RsStMach class contains a state table that indicates the procedures executed for each RsStMach instance. The RsStMach class derives from the RsObject class.

A state is defined by the current state number, 'state,' and a substate number, 'substate.' All state values must be greater than or equal to zero. All state tables must use the zero state as the reset state.

A state table contains one or more state entries. A state entry consists of a state number, a state procedure, and a lock flag indicator. Whenever the current state number matches the

state entry's state number, the corresponding state procedure is executed. The lock flag indicator is used to inhibit the switching from the current state to the requested state.

The public interface of the RsStMach class follows:

```
RsStMach(RsObject* userValue);
virtual ~RsStMach();
void update(const float dt);
int state(void) const;
int substate(void) const;
void reqState(const int newState);
void reset(void);
void setStateTable(StateEntry table[], const int numEntries);
void setBeforeProc(StMachProc procName);
void setAfterProc(StMachProc procName);
float deltaTime(void) const;
void call(const int newState);
void rtn(void);
void jump(const int newState, const int newSubstate = 0);
int next(void);
int goSubstate(const int newSubstate);
int hold(void) const;
int lock(const int flg);
int findState(const int state);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.10 RsSweep

The RsSweep classs stands as a container class that will maintain one longitudinal sweep of data. The longitudinal sweep of data may include terrain elevation posts, terrain culture data, and simulated radar return data. The data points that are maintained are evenly spaced between the minimum and maximum ranges, depending on the number of data points. Range values are measured in meters. Instances of the RsSweep class may or may not be memory managed by the class itself, depending on the constructor used. The RsSweep class derives from the RsObject class.

The public interface of the RsSweep class follows:

```
RsSweep(float* data, const int n,
const float angle, const float minRng, const float maxRng);
RsSweep(const RsSweep& sweep);
RsSweep(const int n);
virtual ~RsSweep();
void operator=(const RsSweep&);
int getNumPts(void) const;
float getAngle(void) const;
float getMinRng(void) const;
float getMaxRng(void) const;
```

```
float getDeltaRng(void) const;
float getRange(const int idx) const;
void setData(float* data, const int n);
void setAngle(const float angle);
void setMinRng(const float minRng);
void setMaxRng(const float maxRng);
float* getDataAddress(void);
const float* getDataAddress(void) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.11 RsTimer

The RsTimer class stands as a general purpose class of up/down timers. The real-time interval timer can be used to update the timers or they can be updated manually by the instance of the RsTimer class. Using instances of the RsTimer class, processes can be suspended until the real-time interval timer sends the SIGALRM signal. Instances of the RsTimer class may count positively or negatively. The RsTimer class derives from the RsObject class.

The public interface of the RsTimer class follows:

```
RsTimer(int direction, float rtime = 0.0f);
virtual ~RsTimer();
float time(void) const;
void start(void);
void stop(void);
void reset(void);
void reset(const float rtime);
void update(const float dt);
int alarm(void) const;
int alarm(const float atime);
static int freeze(const int ff);
int freeze(void) const;
static float getDeltaTime(void);
static void updateTimers(const float dt);
static void setItimer(const float dt);
static void waitOnItimer(void);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.12 RsUpTimer

The RsUpTimer class stands as a general purpose class of magnitude increasing timers. The instances of the RsUpTimer class are timers that count in the positive direction. The RsUpTimer class derives from the RsTimer class.

The public interface of the RsUpTimer class follows:

```
RsUpTimer(float rtime = 0.0f);
virtual ~RsUpTimer();
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.13 RsDownTimer

The RsDownTimer class stands as a general purpose class of magnitude decreasing timers. The instances of the RsDownTimer class are timers that count in the negative direction. The RsDownTimer class derives from the RsTimer class.

The public interface of the RsDownTimer class follows:

```
RsDownTimer(float rtime = 0.0f);
virtual ~RsDownTimer();
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.14 RsGufForm

The RsGufForm class stands as a general GUF form class. Instances of the RsGufForm class are used primarily by the parser when it parses the configuration files. As the parser examines the GUF configuration files, it instantiates the various objects needed for the radar system. The RsGufForm class is a container class that maintains a form name, 'formName,' and a list of arguments, 'argList.' The items that are maintained by the RsGufForm class are references to an instance of the RsString class and an instance of the RsGufPairStream class. For the support of GUF, certain functions are provided within the public interface of the RsGufForm class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The RsGufForm class derives from the RsObject class. The syntax for a general GUF form is as follows:

General Form Syntax

```
( formName <argList> )
```

The public interface of the RsGufForm class follows:

```
RsGufForm(const char* form, RsGufPairStream* list);
RsGufForm(const RsGufForm&);
```

```
virtual ~RsGufForm();
const RsIdent* form(void) const;
RsGufPairStream* argList(void);
const RsGufPairStream* argList(void) const;
RsGufForm& operator=(const RsGufForm&);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
static const RsType type;
```

6.1.15 RsGufPair

The RsGufPair class stands as a general GUF slot pair class. Instances of the RsGufPair class are used primarily by the parser when it examines the configuration files. As the parser examines the GUF configuration files, it instantiates the various objects needed for the radar system. The RsGufPair class is a container class that maintains a slot name and an object. Instances of the RsGufPair class maintain a reference to an instance of the RsString class, the slot name, and a value which is a reference to an instance of the RsObject class. The RsObject class stands as an abstract class that cannot be instantiated. Therefore, instances for the object reference are actually instances of classes derived from the RsObject superclass, such as RsIntegers or RsFloats. The syntax for a general GUF slot pair follows:

General GUF Slot Pair Syntax

```
: slotName <value>
```

The public interface of the RsGufPair class follows:

```
RsGufPair(const char* slot, RsObject* object);
RsGufPair(const RsGufPair&);
virtual ~RsGufPair();
const RsIdent* slot(void) const;
RsObject* object(void);
const RsObject* object(void) const;
RsObject* removeObject(void);
RsGufPair& operator=(const RsGufPair&);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
void print(ostream& sout, const int indent);
static const RsType type;
```

6.1.16 RsGufSlotTable

The RsGufSlotTable class stands as container class that maintains the slot table for each of the GUF defined forms. The slot table is used to assign slot names to indices. If no names are given for the slots, ordinal values are automatically assigned. Each GUF defined form for the radar simulation, mainttains an instance of the RsGufSlotTable class. Two constructors are provided with the RsGufSlotTable class in the event the base table from a base class is needed for the instantiation of the object. Typically, an array of character strings and the number of slots is provided to the contructor. The main purpose of the RsGufSlotTable is to provide a common interface to all of the GUF defined forms to their respective RsGufSlotTable. An index from the table may be obtained given an input index with the index() member function; the slot name can be obtained given an input character string with the name() member function. The number of the last index can be obtained via the n() member function. Finally, the ability to print the information within the RsGufSlotTable is provided via the print() member function.

The public interface of the RsGufPair class follows:

```
RsGufSlotTable(const char* slots[], const int nslots,
            const RsGufSlotTable& baseTable);
RsGufSlotTable(const char* slots[], const int nslots);
~RsGufSlotTable();
int index(const char* slotname) const;
const char* name(const int slotindex) const;
int n(void) const { return lastIndex; }
void print(ostream& sout) const;
```

6.1.17 RsList

The RsList class stands as a general purpose container class that maintains a list of objects. The objects within the list are maintained with a doubly-linked list. Several utilities are available with the RsList class that facilitate the maintenance of instances of the RsList class. The items that are maintained within the RsList class are references to instances of the RsObject class. The RsList class provides the utility to find a specific object within the instance of the RsList class with the find() member function. Also, the RsList class provides the utility to insert before an RsObject within the instance of the RsList class via the insert() member function. In addition, the RsList class provides the utility to replace or remove a specified object within the instance of the RsList class, assuming the object exists within the

list. The RsList class also provides the ability to retrieve an array of RsNumbers or RsGufPairs that contain RsNumbers from a list with the getNumberList() member function. For the support of GUF, certain functions are provided within the public interface of the RsGufForm class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The RsList class derives from the RsObject class.

The public interface of the RsList class follows:

```
RsList(void);
RsList(const float values[], const int nv);
RsList(const int values[], const int nv);
RsList(const RsList& list);
virtual ~RsList(void);
int operator==(const RsList& list) const;
int operator!=(const RsList& list) const;
RsList& operator=(const RsList& list);
int isEmpty(void) const;
int entries(void) const;
RsObject* get(void);
void put(RsObject* obj);
int getNumberList(float values[], int max);
RsObject* first(void);
RsObject* next(void);
RsObject* previous(void);
RsObject* position(const int n);
RsObject* last(void);
int find(RsObject* obj);
int insert(RsObject* obj);
RsObject* replace(RsObject* obj);
int remove(RsObject* obj = NULL);
// Routines that handle the list's head and tail.
void addHead(RsObject* obj);
void addTail(RsObject* obj);
RsObject* removeHead(void);
RsObject* removeTail(void);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.1.18 RsStack

The RsStack class stands as a general purpose container class that maintains a stack of objects. The objects within the stack are maintained with a doubly-linked list in the Last In First Out (LIFO) fashion. The RsStack class derives from the RsList class publicly which

means that in addition to the common push and pop utilities, all of the RsList utilities are also available to instances of the RsStack class.

The public interface of the RsStack class follows:

```
RsStack(void);
RsStack(const RsStack& stack);
virtual ~RsStack(void);
int operator==(const RsStack& list) const;
int operator!=(const RsStack& list) const;
RsStack& operator=(const RsStack& list);
RsList::isEmpty;
RsList::entries;
RsList::first;
RsList::next;
void push(RsObject* object);
RsObject* pop(void);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.19 RsGufPairStream

The RsGufPairStream class stands as a container class that maintains a stream of GUF pairs, GUF form lists as discussed in *Section 4, Software Design Approach.* Instances of the RsGufPairStream are primarily used by the parser. The RsGufPairStream class is a container class that maintains a stream of RsGufPair instance references. The RsGufPairStream class derives privately from the RsList class, which means the underlying implementation of the RsGufPairStream is also a doubly-linked list of RsGufPairs. The formats available for the RsGufPairStream is as follows:

Available RsGufPairStream Formats

```
      ( guf <gufPairStream> )
or
      '( <gufPairStream> )
or
      <argList>
```

The public interface of the RsGufPairStream class follows:

```
RsGufPairStream(void);
RsGufPairStream(const RsGufPairStream& stream);
virtual ~RsGufPairStream(void);
int operator==(const RsGufPairStream& stream) const;
int operator!=(const RsGufPairStream& stream) const;
RsGufPairStream& operator=(const RsGufPairStream& stream);
```

```
RsList::isEmpty;
RsList::entries;
RsList::getSlotByIndex;
RsGufPair* first(void);
RsGufPair* next(void);
RsGufPair* previous(void);
RsGufPair* last(void);
RsGufPair* position(const int n);
RsGufPair* get(void);
void put(RsGufPair* pair);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const RsType type;
```

6.1.20 RsColor

The RsColor class stands as an abstract class of colors. This class defines the colors to consist of four floating point values: red, green, blue, and alpha. The alpha value is set to a default value of 1.0. The RsColor class derives from the RsObject class. The Performer math include files are located within this class because the colors are handled with the three, pfVec3, and four, pfVec4, component vectors, which are defined within these include files. To comply with the data encapsulation feature of object-oriented code, the following functions provide an interface to obtain the components of an RsColor object: red(), green(), blue(), alpha(), getRGB() and getRGBA(). The RsColor class derives from the RsObject class.

The public interface of the RsColor class follows:

```
RsColor(void);
virtual ~RsColor();
operator const pfVec3*(void) const;
operator const pfVec4*(void) const;
float red(void) const;
float green(void) const;
float blue(void) const;
float alpha(void) const;
void getRGB(pfVec3 rgb) const;
void getRGBA(pfVec4 rgb) const;
static void setDefaultAlpha(const float alpha);
virtual int isClassType(RsType type) const;
static const char* form;
static const RsType type;
```

6.1.21 RsRgb

The RsRgb class stands as a class that defines a color to contain a red, a green, and a blue component with no specification for the alpha component. The main addition the RsRgb class provides, which is not provided by the RsColor class, is the support for the GUF slot table. The addition of the slot table indicates that the RsRgb class is actually a GUF form. For the support of GUF, certain functions are provided within the public interface of the RsRgb class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The RsRgb class derives from the RsColor class. There are three components within RsRgb's GUF slot table (see Table 5). The values of the red, green, and blue slots, in the range of 0.0 to 1.0, determine a specific color.

Table 5. RsRgb's GUF Slots

Slot Index	Slot Name	Value Type	(Range)
1	red	float	(0.0 - 1.0)
2	green	double	(0.0 - 1.0)
3	blue	float	(0.0 - 1.0)

The public interface of the RsRgb class follows:

```
RsRgb(const float r, const float g, const float b);
RsRgb(void);
virtual ~RsRgb();
friend int operator==(const RsRgb&, const RsRgb&);
friend int operator!=(const RsRgb&, const RsRgb&);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual void print(ostream& sout,const int indent = 0,
          const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.1.22 RsRgba

The RsRgba class stands as a class that defines a color to contain a red, a green, a blue, and an alpha component. The RsRgba class derives publicly from the RsRgb class. The slot table inherited from the RsRgb class indicates the RsRgba class is also a GUF form. For the support of GUF, certain functions are provided within the public interface of the RsRgba class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The

main addition of the RsRgba class is that the GUF slot table has an additional slot for the alpha value for each instance of the RsRgb class. Therefore, there are four components within the slot table (see Table 6). The values of the red, green, blue, and alpha slots, in the range of 0.0 to 1.0, determine a specific color.

Table 6. RsRgba's GUF Slots

Slot Index	Slot Name	Value Type (Range)	
1	red	float	(0.0 - 1.0)
2	green	double	(0.0 - 1.0)
3	blue	float	(0.0 - 1.0)
4	alpha	float	(0.0 - 1.0)

The public interface of the RsRgba class follows:

```
RsRgba(const float r, const float g, const float b, const float a);
RsRgba(void);
virtual ~RsRgba();
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual void print(ostream& sout,const int indent = 0,
          const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.1.23 RsHsv

The RsHsv class stands as a class that defines a color to contain a hue, a saturation, and a value component without an alpha component. The main addition the RsHsv class provides, which is not provided by the RsColor class, is the support for the GUF slot table. The addition of the GUF slot table indicates that the RsHsv class is actually a GUF form. For the support of GUF, certain functions are provided within the public interface of the RsHsv class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The RsHsv class derives from the RsColor class. There are three components within RsHsv's GUF slot table (see Table 7). The combined values of the hue, saturation, and value slots determine a specific color.

For documentation purposes, the hue slot represents a degree around a color wheel with a maximum value of 360° and a minimum value of 0°. The saturation slot represents whether the hue is a pure hue with a value of 1.0, and the value slot represents whether the intensity/tint is at a maximum value of 1.0. Example hues, saturation, and values are provided in Table 8.

Table 7. RsHsv's GUF Slots

Slot Index	Slot Name	Value Type	(Range)
1	hue	float	(0.0 - 360.0)
2	saturation	double	(0.0 - 1.0)
3	value	float	(0.0 - 1.0)

The public interface of the RsHsv class follows:

```
RsHsv(const float h, const float s, const float v);
RsHsv(void);
virtual ~RsHsv();
float hue(void) const;
float saturation(void) const;
float value(void) const;
void getHSV(pfVec3 hsv) const;
friend int operator==(const RsHsv&, const RsHsv&);
friend int operator!=(const RsHsv&, const RsHsv&);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
friend void hsv2rgb(pfVec4 rgb, const pfVec4 hsv);
static const char* form;
static const RsType type;
```

6.1.24 RsHsva

The RsHsva class stands as a class that defines a color to contain a hue, a saturation, a value and an alpha component. The RsHsva class derives publicly from the RsHsv class. The slot table inherited from the RsHsv class indicates the RsHsva class is also a GUF form. For the support of GUF, certain functions are provided within the public interface of the RsHsva class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The main addition of the RsHsva class is that the GUF slot table has an additional slot for the

alpha value for each instance of the RsHsv class. Therefore, there are four components within the slot table (see Table 9).

Table 8. Example Colors Using Hue, Saturation, and Value Components

Color	Hue
red	0.0
yellow	60.0
green	120.0
cyan	180.0
blue	240.0
magenta	300.0
red	360.0
	Saturation
maximum hue	1.0
no hue (white)	0.0
	Value
maximum intensity	1.0
no intensity (black)	0.0

The values of the saturation, value, and alpha components in the range of 0.0 to 1.0 determine a specific color range from 0.0 to 1.0. The values of the hue component in the range of 0.0 to 360.0 also facilitate in the determination of a color. For documentation purposes, the hue slot represents a degree around a color wheel. The saturation slot represents whether the hue is a pure hue with a maximum value of 1.0. The value slot represents whether the intensity or tint is at a maximum value or 1.0. Example colors are represented within *Table 8, Example Colors Using Hue, Saturation, and Value Components.*

Table 9. RsHsva's GUF Slots

Slot Index	Slot Name	Value Type	(Range)
1	hue	float	(0.0 - 360.0)
2	saturation	double	(0.0 - 1.0)
3	value	float	(0.0 - 1.0)
4	alpha	float	(0.0 - 1.0)

The public interface of the RsHsva class follows:

```
RsHsva(const float h, const float s, const float v, const float a);
RsHsva(void);
virtual ~RsHsva();
void getHSVA(pfVec4 hsva) const;
```

```
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
          const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.1.25 RsClip3D

The RsClip3D class provides the functionality to clip three-dimensional (3D) polygons with respect to a 3D clipping volume. The RsClip3D class derives from the RsObject class. The public interface to this class provides the capability to set the 3D clipping volume with (x,y,z) coordinates or only (x,y) coordinates via the SetBox() member functions. In the later case, the Z clipping plane is defaulted to Z = - FLT_MAX and Z = FLT_MAX. FLT_MAX is the constant stored in the <limits.h> file, which represents the maximum decimal value of a float as defined by Version 5.2 of the IRIX operating system. Two interfaces exist to actually perform the 3D clipping through the member functions entitled clip(). One of these interfaces uses the pfVec3 data structure defined by Iris Performer to identify the polygon's input vertices and the polygon's output vertices after the clipping. The other interface uses the RsPolygon reference to identify the polygon's input and output vertices. In both of these interfaces, the clip() method identifies the number of vertices remaining in the clipped polygon output. If the polygon exists outside of the clipping volume, the number of vertices returned through either interface is zero.

The public interface of the RsClip3D class follows:

```
RsClip3D(void);
virtual ~RsClip3D();
int clip(RsPolygon& p);
int clip(pfVec3 out[], const pfVec3 in[], const int n);
int clip(pfVec3 vout[], pfVec3 nout[],
const pfVec3 v[], const pfVec3 vn[], const int tn);
void setBox(const float xmin, const float xmax,
const float ymin, const float ymax,
const float zmin = -FLT_MAX, const float zmax = FLT_MAX);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.26 RsFlatEarth

The RsFlatEarth class provides simple flat earth conversion from the center of the terrain database. The RsFlatEarth class derives from the RsObject class, and as all derived classes, has the opportunity to use the public and private member functions of the base class. Once the reference center of the database is set in terms of a latitude and longitude with the RsFlatEarth class constructor, the RsFlatEarth class uses conversion constants defined in the support.h file in addition to the Iris Performer data structures, pfVec2 and pfVec3, to calculate specific items of information. Latitude and longitudes are described in degrees, altitude is expressed in feet, and (x,y,z) coordinates are expressed in meters, using the Iris Performer structures pfVec3 and pfVec2. Several interfaces exist publicly within the RsFlatEarth class to convert from a position in latitude, longitude, and altitude to a two or 3D position in (x,y,z) coordinates or vice-versa using the following member functions or methods: lla2vec(), ll2vec(),vec2lla(), and xy2ll(). The RsFlatEarth class also supports the computation of the Euler angles, heading, pitch, and roll and the computation of distances between points via the member functions viewAngles() and distance(). Two additional member functions provide the ability to set the conversion constants for the mathematical conversions from nautical miles (nm) to meters, setNM2M(), and from nms to feet, setFT2M(). The default mathematical conversion constants that are used are defined within the support.h include file, as defined by MultiGen V14.1.

The public interface of the RsFlatEarth class follows:

```
RsFlatEarth(const double lat, const double lon);
virtual ~RsFlatEarth();
void lla2vec(const double lat, const double lon, const double alt,
             pfVec3& v) const;
void ll2vec(const double lat, const double lon, pfVec2& v) const;
void vec2lla(const pfVec3 v, double& lat, double& lon, double& alt) const;
void xy2ll(const pfVec2 v, double& lat, double& lon) const;
void viewAngles(const pfVec3 p1, const pfVec3 p2, pfVec3 view) const;
float distance(const pfVec3 p1, const pfVec3 p2) const;
void setNM2M(const double value);
void setFT2M(const double value);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.1.27 RsPolygon

The RsPolygon class provides the ability to store a 3D polygon. The 3D polygons are stored in terms of the vertices that constitute the polygon. The maximum number of vertices per each object of the RsPolygon class currently stands as ten. The RsPolygon class supports the calculation of the normals at each vertex. In addition, the RsPolygon class supports the calculation of the polygon's surface normal. The RsPolygon class also supports the calculation of the coefficients of the planar equation for each polygon. The implementation assumes that each polygon is planar with respect to 3D space. A color is also associated with the RsPolygon class. A layer value is also an attribute to each polygon of the RsPolygon class. The RsPolygon class derives from the RsObject class. The RsPolygon class is 'friends' with the RsClip3D class.

The public interface of the RsPolygon class follows:

```
enum { a=0, b=1, c=2, d=3 };
enum { vertMax = 10 };// maximum number of vertices
RsPolygon(void);
RsPolygon(const RsPolygon&);
virtual ~RsPolygon();
void operator=(const RsPolygon&);
void import(const pfVec3 v[], const pfVec3 vn[], const int n);
void import(const pfVec3 v[], const int n);
int export(pfVec3 v[]) const;
int exportNorms(pfVec3 vn[]) const;
void getColor(pfVec4 rgba) const;
void setColor(const pfVec4 rgba);
int getLayer(void) const;
void setLayer(const int newLayer);
void getNormal(pfVec4 norm) const;
void setNormal(const pfVec4 norm);
void calcNormal(void);
static void calcNormal(pfVec3 norm, const pfVec3 vert[3]);
void calcPlaneCoeff(void);
static void calcPlaneCoeff(pfVec4 coeff, const pfVec3 vert[3]);
float calcZ(const pfVec2 point) const;
static float calcZ(const pfVec2 point, const pfVec4 coeff);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const RsType type;
friend class RsClip3D;
```

6.1.28 RsScanline

The RsScanline class provides the ability to draw a set of polygon objects in 3D space within a 2D display area. The algorithm that is used to process the drawing of the active polygon set

is called the scan-conversion algorithm. The scan-conversion algorithm operates on the vertices of the polygons. The basic strategy behind this algorithm determines which pixels within each single scan line are to be filled within the polygon and set the pixel to the corresponding value. This strategy is repeated for each scan line and for each polygon with several efficient steps to scan-convert the entire set of polygons (Foley, J.D. and Van Dam, A. 1982). The RsScanline class derives from the RsObject class. The RsScanline class uses a class internally, called the Polygon class. The internal Polygon class is derived from the RsPolygon class. In addition, the RsScanline class defines and uses a structure for the edges of the polygon for efficiency.

The public interface of the RsScanline class follows:

```
RsScanline(void);
virtual ~RsScanline();
void area(const float x, const float y,const float dx,
          const float dy, const float a);
void size(const int x, const int y);
void addPolygon(const pfVec3 v[], const int n, const pfVec3 pn,
          const pfVec4 color, const int layer = 0);
void addPolygon(const pfVec3 v[], const pfVec3 vn[], const int n,
          const pfVec4 color, const int layer = 0);
void setCallback(RsScanlineCallbackProc proc, RsObject* userData);
void erase(void);
void reset(void);
const Polygon* step(const int x);
void scanline(const int y);
void scan(void);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
static const RsType type;
```

6.2 Parser Library

The Parser library, denoted as libParser, primarily supports the input configuration files. For a complete discussion of the APG-70 parser data flow, please review *Section 4, Software Design Approach.* As discussed within Section 4, the parser consists of two public domain software components, Bison++ and Flex++.

Bison++ is a general purpose parser generator that converts a context-free grammar into a C++ member function that parses the respective grammar. Within the Parser Library, the Bison++ context-free grammar exists within the file named parser.y. The corresponding C++ file created by Bison++ follows as parser.c++.

Flex++ is a fast lexical analyzer generator compatible with Bison++. Flex++ uses an input file containing a context-free grammar that dictates the lexical tokens it should recognize. The input file within the Parser Library is named lexical.l. Flex++ converts lexical.l into a C++ file, lexical.c++, which contains a C++ member function to work with the parse() function defined within parser.c++.

In addition to these two components, each class which can be instantiated via the input configuration files, must contain a formFunc() member function which simply instantiates the class externally. These member functions within the radar simulation are commonly found within the formfunc.c++ files throughout the yars directories. Since no other classes are defined by the Parser Library, no class hierarchy exists.

6.3 Radar Library

The Radar library, denoted as libRadar, primarily supports the common objects and subsystems needed for the different modes of the radar model, such as the stand by mode, the off mode, the real beam map (RBM) plan position indicator (PPI) mode, and the high resolution map (HRM) patch map mode. The class hierarchy for the objects which are derived from the RsRadar class is depicted in Figure 11 using Booch's notation (Booch, G 1994). This class hierarchy shows the inheritance relationships among the various classes within the radar library in addition to classes which exist in other libraries. The majority of objects within the radar system code are derived from the RsObject class which exists within the Basic library.

The major components of the Radar library are the RsDatabase, RsAntenna, RsMode, and RsDisplay subclasses. These subclasses of the RsRadar class primarily contain the functionality and information encapsulation common among the various radar subsystems. To support a simulated radar system, a database of terrain information must exist so that the radar system can simulate the scanning of the terrain. The RsDatabase class provides the accessibility to terrain information. To drive the positioning of the radar scan, a physical antenna is simulated. The RsAntenna class provides the simulation of the various

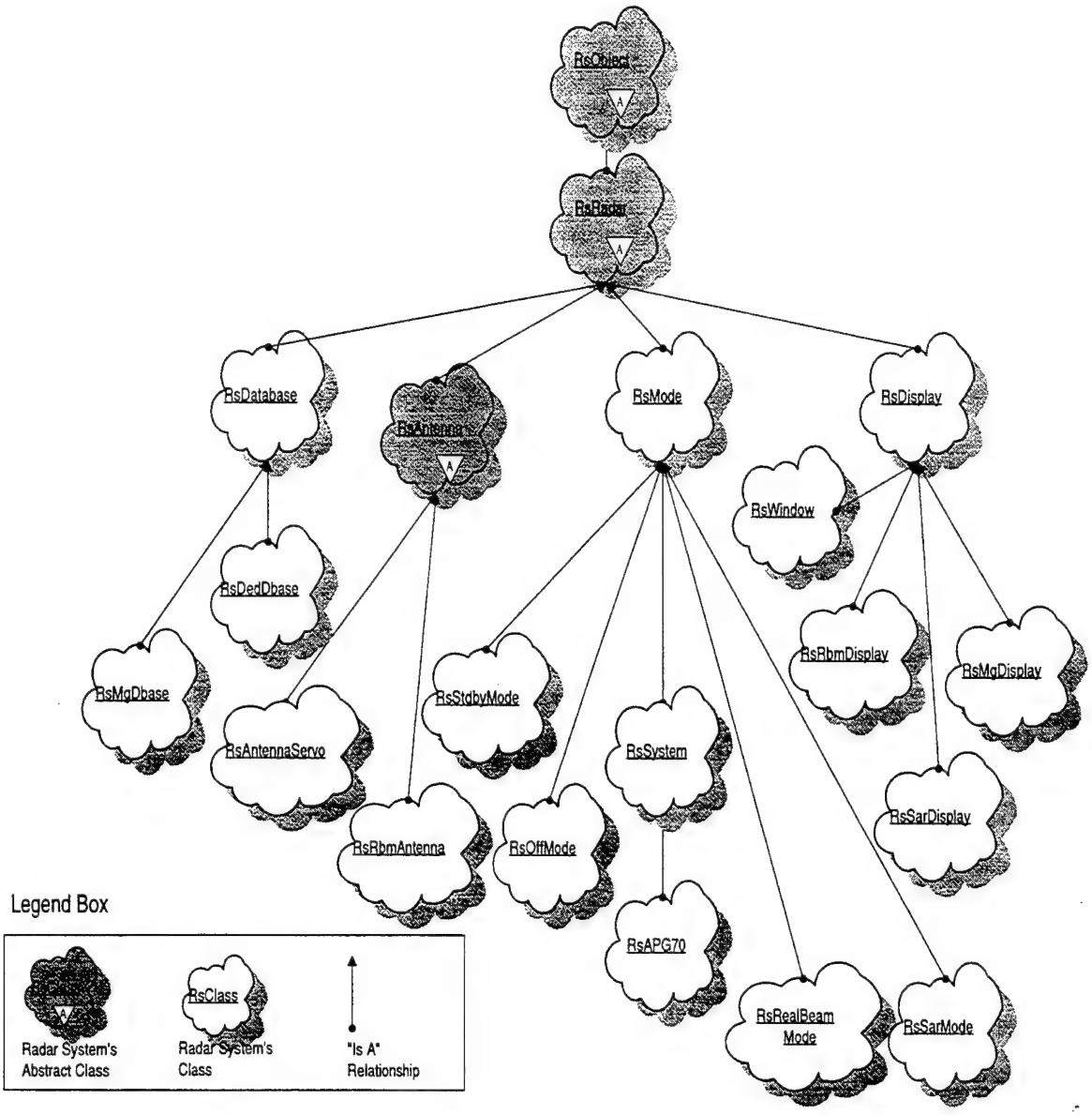


Figure 12. Class Hierarchy for RsRadar Derived Subclasses

components needed for a physical antenna. As a radar system has a set of modes or system states in which it operates, the RsMode class supports the common mode functions and data for the radar simulation. In certain modes, such as the RBM PPI, the antenna scans the terrain and outputs the terrain information on a display which is viewed by the crew in the aircraft. Within the simulation environment, the simulated antenna scans the terrain database and calculates the display output dependent on the location of the radar beam source. To

support the displaying of the simulated radar output, return sweeps, the RsDisplay class was created.

As the radar operates in the various modes, radar system parameters may change causing the radar system to switch modes or to change antenna position or even change the accessible terrain information due to the radar range and the position of the aircraft. The derived subclasses from the RsDatabase, RsAntenna, RsMode, and RsDisplay classes provide the various specific features needed to fully simulate the F-15E's APG-70 radar system with regard to the RBM PPI mode and the HRM patch map mode. If additional modes of the aircraft's radar system were added to the current simulation (e.g., air-to-air, precision velocity update, or beacon modes) new classes would need to be created within the radar library to simulate the specific radar system functionality dictated by the modes.

To enhance the understanding of the Radar library, each individual class and its respective public interface is explained in the following paragraphs. In addition, the following diagram indicates the classes consitituting the Radar library (Figure 13).

6.3.1 RsRadar

The RsRadar class stands as an abstract base class primarily for the radar subsystems, such as the RsDatabase, RsAntenna, RsMode, and RsDisplay derived subclasses. The RsRadar class derives from the RsObject class. The major responsibility of the RsRadar class is to initialize the static data before any classes are created. The initialization routine, init(), is passed a void pointer. If the void pointer points to a shared memory area, the size of the shared memory is given in bytes. If no arguments are provided to the initialization routine, local

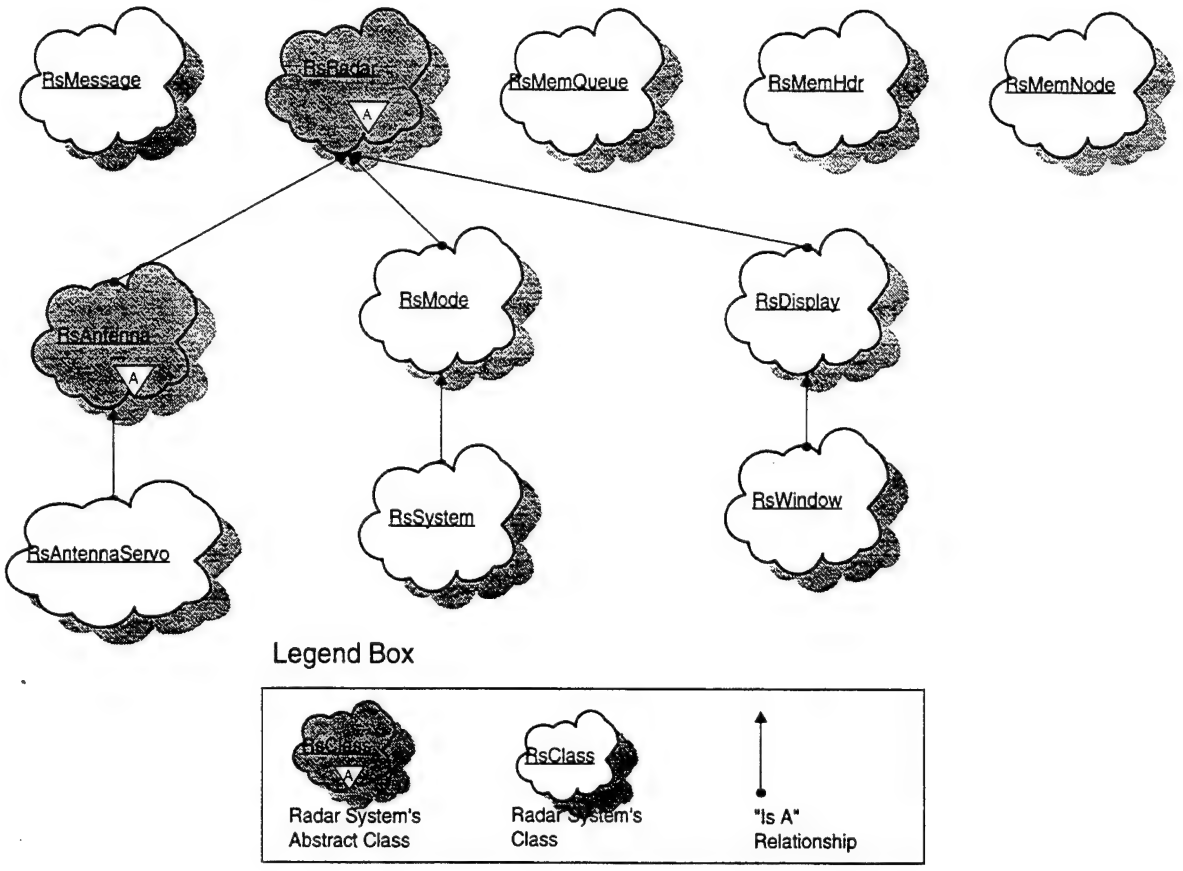


Figure 13. Radar Library's Classes

memory is allocated for the static data area. The reference to the static data area is actually a display queue that is used to pass radar image data using the RsSweep class, to the radar display routines. Other key member functions contained within this class are the setup() and the update() member functions. These functions are defined as purely virtual member functions indicating that the derived subclasses for the RsRadar class must define these member functions respectively. This configuration permits run-time dynamic binding and confirms flexibility of the radar simulation. The setup() member function is called after an object has been instantiated and all of the Grand Unified File (GUF) slots have been set by the parser. The setup() member function provides a method to notify radar components that they may setup internal data. The update() member function is dependent upon the derived subclass, as named, this class updates the radar subsystem information. The RsRadar class also defines a function pointer type RsCallbackProc that has two parameters of pointers to an RsRadar object and a RsObject object. This type, RsCallbackProc, is used to perform the

function callbacks for the associated update routines within the correct RsRadar derived subclasses.

The public interface of the RsRadar class follows:

```
typedef void (*RsCallbackProc)(RsRadar* obj, RsObject* userData);
RsRadar(void);
virtual ~RsRadar();
virtual void update(const float dt) = 0;
virtual void setup(RsRadar* owner) = 0;
static RsMemQueue* displayQueue;
virtual int isClassType(RsType type) const;
static const RsType type;
static void init(void);
static void init(void* mem, const int memsize = 0);
```

6.3.2 RsAntenna

The RsAntenna class stands as an abstract class primarily functioning as a data store for its derived subclasses. The RsAntenna class stores significant information regarding the state of the simulation of an antenna, such as the current antenna position (degrees), the current velocity (degrees/second), the commanded antenna position (degrees), and the commanded antenna rate (degrees/second). Other information that is also encapsulated in the RsAntenna abstract class are common antenna parameters. The maximum slew rate (degrees/second), the left/lower and right/upper gimbal limits (degrees/second), the horizontal beam width and vertical beam height (degrees), and the antenna reference angle (degrees) exist as the primary antenna parameters. The RsAntenna class is derived publicly from the RsRadar class, which means all of the public members from the RsRadar class are also public members to the RsAntenna class. Three antenna servo modes, position, rate, and frozen, are enumerated within the RsAntenna class to facilitate the simulation of a moving antenna. The RsAntenna class uses the Iris Performer math data structures to store vectors associated with the positioning of the radar antenna.

The primary function of the methods specific to the RsAntenna class are to encapsulate the current antenna parameters such as the position angles, the angular rates, the reference angles, and the horizontal and vertical beam sizes. The antenna reference angle parameter measured in degrees is the direction for the antenna to scan, the center of the scan pattern. A few other

members of the class provide the ability to toggle the antenna beam on or off, turnBeamOn() or turnBeamOff(), to indicate whether the antenna is off or is in an aircraft shadow, isBlanked(), and to indicate whether the antenna is positioned within a specified tolerance of the commanded antenna position, isPositioned(), or to indicate whether the antenna is at one of the physical limits, isAtLimits(). A default tolerance of 0.1 degrees is used when no tolerance is given. As this is an abstract class, a few protected member functions are identified to set the antenna servo mode, to set the commanded antenna rates, and to set the commanded antenna position. Since these member functions are protected, only the derived subclasses of the RsAntenna class have the ability to use or redefine these member functions. In addition, two vector limiting functions are accessible through the RsAntenna class.

The public interface of the RsAntenna class follows:

```
enum ServoMode { freezeServo, rateServo, positionServo };
RsAntenna();
virtual ~RsAntenna();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
void getPosition(pfVec2& position) const;
float getAzimuth(void) const;
float getElevation(void) const;
void getRates(pfVec2& rates) const;
float getAzimuthRate(void) const;
float getElevationRate(void) const;
void getRefPosition(pfVec2& refPosition) const;
float getRefAzimuth(void) const;
float getRefElevation(void) const;
float horzBeamWidth(void) const;
float vertBeamWidth(void) const;
static void turnBeamOn(void);
static void turnBeamOff(void);
int isBlanked(void) const;
int isPositioned(const float tol = defaultAntennaTolerance) const;
int isAtLimits(void) const;
friend int limitVec(pfVec2 vec, const pfVec2 lim);
friend int limitVec(pfVec2 vec, const pfVec2 ll, const pfVec2 ul);
void setRefPosition(const pfVec2 refPosition);
void setRefAzimuth(const float refAzimuth);
void setRefElevation(const float refElevation);
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
          const int slotsOnly = FALSE);
static const RsType type;
```

6.3.3 RsAntennaServo

The RsAntennaServo class stands as a representative for a physical antenna simulation. The RsAntennaServo class uses the Iris Performer math data structures to store vectors associated with the positioning of the radar antenna. The RsAntennaServo class is a derived subclass of the RsAntenna class, which means that all of the public and protected member functions are accessible to the RsAntennaServo class. The setup() member function must be defined by the RsAntennaServo class because the RsAntenna class defines it as a virtual member function. The setup() function specific to the RsAntennaServo class is called by the owner of the RsAntennaServo object that was instantiated. The owner of the RsAntennaServo class will know when the static data for the radar system has been initialized, and following this event, the respective setup() member function of the RsAntennaServo class will be called. The member function update() of the RsAntennaServo class serves to update the data encapsulated by the RsAntennaServo that may cause the radar sweep data to be changed if the radar system is in an active scanning mode.

The RsAntennaServo is one of the classes that is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsAntennaServo class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The RsAntennaServo class uses the Iris Performer math data structures to store vectors associated with the positioning of the radar antenna. As the RsAntennaServo class is publicly derived from the RsAntenna class, the slot names that are supported by the RsAntennaServo class represent the data encapsulated by the RsAntenna class. Therefore, the slots of the RsAntennaServo class store significant information regarding the physical limits of the instantiated RsAntennaServo, such as the upper and lower azimuth limits, the upper and lower elevation limits, the azimuth and elevation rate limits, the horizontal and vertical beam sizes, and the reference position of the antenna measured in azimuth and elevation. The syntax for the RsAntennaServo GUF defined form follows the slot table for the RsAntennaServo GUF defined form:

Table 10. RsAntennaServo's GUF Slots

Slot Index	Slot Name	Value Type
1	azimuthLimits	float[2] (upper & lower) azimuth limits
2	elevationLimits	float[2] (upper & lower) elevation limits
3	maxRates	float[2] azimuth & elevation rate limits
4	beamWidth	float[2] horizontal & vertical beamwidth
5	position	float [2] azimuth & elevation position

```
( def-form AntennaServo
      'azimuthLimits
      'elevationLimits
      'maxRates
      'beamWidth
      'position
)
```

An example of instantiating the physical antenna model within the input configuration file follows:

```
;; Physical Antenna model example
:antenna     ( AntennaServo
      :azimuthLimits     '(-60.0 60.0)
      :elevationLimits   '(-60.0 60.0)
      :maxRates          '(60.0 60.0)
      :beamWidth         '(2.5 2.5)
            )
```

The public interface of the RsAntennaServo class follows:

```
RsAntennaServo();
virtual ~RsAntennaServo();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.3.4 RsDisplay

The RsDisplay class stands as a representative for an output display device. The RsDisplay class is a derived subclass of the RsRadar class, meaning that it must define the virtual member functions: setup(), and update(). The setup() member function of the RsDisplay

class actually creates the display for the owner of the instantiated object. Again, as with the RsAntennaServo class, the owner of the RsDisplay class knows exactly when the static data area is initialized and calls the RsDisplay's setup() following the initialization. The update() member function uses the components that are setup for the RsDisplay to display the output of the radar sweep data. Each instance of the RsDisplay class defines a display model that is associated with a radar mode, respective owner. The basic mode (RsDisplay) clears the display background color, which is useful for the off and stand by modes of the radar system.

Other modes that require more complex radar display modes use a derived subclass of the RsDisplay class. For clarification, the derived subclass, RsWindow, is used to manage the static member variables that define the Iris Graphics Library (GL) graphics window. The information that is encapsulated within the RsDisplay class stores information such as whether or not the RsDisplay is in double-buffered, isDoubleBuffer(), or single-buffered mode, isSingleBuffer(), the pixel width and height of the RsDisplay, the unique Iris GL handle that is returned for each Iris GL window, getWindowId(), the intensity scaling, and the base list of colormap entries. Several member functions are provided to determine the graphical information regarding the Iris GL window. If a window has been created within the RsDisplay object, the function haveWindow() returns TRUE. If the RsDisplay is in color map mode or in red green blue (RGB) mode, the respective functions isColormapMode() and isRGBMode() return TRUE. If the RsDisplay is in color map mode, setting and getting the color map intensity is provided with the setIntensity() and getIntensity() member functions. To clear the RsDisplay, the erase() member function can be called.

Table 11. RsDisplay's GUF Slots

SlotIndex	Slot Name	Value Type
1	width	float; defines left/right ortho2()
2	height	float, defines bottom/top ortho2()

The RsDisplay class is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsDisplay class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The GUF slots of

the RsDisplay class's slot table define the virtual size (ortho2) of the display (see Table 11). The syntax of the RsDisplay defined form follows:

```
( def-form Display
      'width
      'height
)
```

An example of instantiating two objects of the RsDisplay class which both belong to a mode within the GUF-based input configuration file follows:

Example

```
:modes   '(
            (Display
            ;; ortho2 size of virtual display
                  :width   1.0
                  :height  1.0
            )
            (Display
            ;; ortho2 size of virtual display
                  :width   1.0
                  :height  1.0
            )
         )
```

The public interface of the RsDisplay class follows:

```
const long MAP_MAX_COLORS = 256;
RsDisplay();
virtual ~RsDisplay()
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void redraw(void);
static int haveWindow(void);
static int isDoubleBuffer(void);
static int isSingleBuffer(void);
static int isColormapMode(void);
static int isRGBMode(void);
static int getWinId(void);
static void erase(void);
static float getIntensity(void);
static void setIntensity(const float intensity);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
friend RsWindow;
```

6.3.5 RsMessage

An early version of the radar system used messages as the primary method to pass data from the message-based configuration files to the radar component classes. In the current radar simulation, the RsMessage is used only by RsMemQueue to pass data messages via shared memory. The RsMessage communication within the first version of the radar system primarily was based on keywords and pointers (references) to specific instantiated objects. Based on its type, the RsMessage can contain a number, a string, a void pointer, a RsSweep, or a linked-list. A RsMessage's type is determined when it is constructed and can not be changed. A RsMessage's type and respective keyword are encapsulated within the RsMessage class and can be determined by a few member functions as indicated by several member functions within the public interface for the RsMessage class. Several constructors are provided to support the wide variety of types of messages, as is noted in the publically defined enumeration Type for the RsMessage class. Comments are provided within the public interface to explain the slight differences among the member function signatures.

In addition, several functions are provided to obtain the encapsulated data within the RsMessage. The member function getNumber() retrieves a number from the RsMessage. The member function, getNumberList() retrieves an array of numbers from a RsList of RsNumbers within the RsMessage object. Additional member functions provide the ability to obtain the RsString, getString(), the void pointer, getPointer(), the RsSweep, getSweep() and the first, getFirst(), or next, getNext() RsMessage of a RsList. Support to manage the keyword message are also provided with the following member functions: hasKeyword(), isKeyword(), getKeyword(), and setKeyword().

The public interface for the RsMessage class follows:

```
enum Type { rsNumber, rsString, rsPointer, rsSweep, rsList };
RsMessage(float value, const char* kw = NULL);
//    Constructor for a rsList type message that will contain 'nv'
//    messages, each of rsNumber type, which contain the values
RsMessage(char* string, const char* kw = NULL);
//    RsMessage(char* string)
//    RsMessage(char* string, char* key)
//    Constructors for a rsString type message.  The pointer to the
//    original string is contained in the message and this string is
//    not deleted with the RsMessage destructor.
```

```
RsMessage(void* ptr, const char* kw = NULL);
//    Constructors for a rsPointer type message.  The pointer to the
//    original data is contained in the message and this data is not
//    deleted with the RsMessage destructor.
RsMessage(RsSweep* sweep, const char* kw = NULL);
//    Constructors for a rsSweep type message.  The pointer to the
//    original sweep is contained by the message and this sweep is not
//    deleted with the RsMessage destructor.
RsMessage(RsMessage* list, const char* kw = NULL);
//    Constructors for a rsList type message.  When this message is
//    deleted, all children messages contained within the list are
//    deleted.  However, if the child message is of rsString, rsSweep,
//    or rsPointer type, its data is not deleted.
RsMessage(const float values[], const int nv, const char* kw = NULL);
virtual ~RsMessage();
Type getType(void) const;
int isNumber(void) const;
int isString(void) const;
int isPointer(void) const;
int isSweep(void) const;
int isList(void) const;
float getNumber(void) const;
int getNumberList(float values[], int max) const;
char* getString(void) const;
void* getPointer(void) const;
RsSweep* getSweep(void) const;
RsMessage* getFirst(void) const;
RsMessage* getNext(void) const;
void add(RsMessage* msg);
int hasKeyword(void) const;
int isKeyword(const char* key) const;
const char* getKeyword(void) const;
void setKeyword(const char* key);
```

6.3.6 RsMode

The RsMode class stands as an abstract mode class, or superclass, for the various radar system modes. The RsMode class is a derived subclass of the RsRadar class, therefore the update() and setup() member functions may be redefined. However, in most cases these member functions are actually redefined at a lower level with derived subclasses of the RsMode class, such as the RsStdbyMode, RsOffMode, RsRealBeamMode, or RsSarMode classes.

The primary function of the RsMode class is to provide a container for the various components of the radar mode such as the display, the antenna, and the terrain databases. Several member functions retrieve pointers of the RsAntenna, the RsDisplay, and the RsDatabase. In addition, the number of samples per sweep is encapsulated within the

RsMode class. In addition, the RsMode class contains memory management member functions that support the RsSweep object.

The RsMode class is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsMode class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The syntax for the RsMode defined form follows the GUF slot table for RsMode:

Table 12. RsMode's GUF Slots

Slot Index	Slot Name	Value Type
1	display	Display model used by mode
2	antenna	Antenna model used by mode
3	databases	Database used by mode
4	samples	Integer; # of samples per sweep

```
( def-form Mode
      'display
      'antenna
      'databases
      'samples
)
```

The public interface of the RsMode class follows:

```
RsMode(void);
virtual ~RsMode();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void redraw(void);
int getSamples(void) const;
RsAntenna* getAntenna(void);
const RsAntenna* getAntenna(void) const;
RsDisplay* getDisplay(void);
const RsDisplay* getDisplay(void) const;
RsDatabase* getDatabase(void);
const RsDatabase* getDatabase(void) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.3.7 RsSystem

The RsSystem class stands as a basic radar system class. The RsSystem class is a derived subclass of the RsMode class, which means that it also is a container class that holds the antenna, the display window, and the terrain database. In addition, the RsSystem class provides mode selection, and has access to static shared memory that is used to pass radar image data among the radar subsystem components. One of the primary member functions provided by this class is createSystem() as it takes a configuration file's path name as input and the parser constructor function and an optional graphics window handle and creates a display system. Once the createSystem() member function has been completed, the RsSystem object simply manages the various system parameters like the current system mode. When the radar system changes mode, the MODE_CHANGED event is fired. The information that is encapsulated within the RsSystem class are the simulation update rate, the flag which indicates that the simulation is running, the valid list of radar modes available, the current radar mode selected, the list of selectable ranges, the currently selected range, the application and graphics callback routines, respective user data, the aircraft position (in terms of latitude, longitude, altitude, and heading), and the point of interest (in terms of latitude, longitude) on the terrain. Event handling via the Iris devices and callback routines are also provided by the RsSystem class. Currently, the maximum number of callbacks is ten.

The RsSystem class is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsSystem class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). Since the RsSytem class is derived from the RsMode class, the RsSystem class inherits the characteristics of the RsMode class (see Table 13). The defined form for the RsSystem class indicates this relationship and follows respectively:

```
( def-form System
      is= Mode     ;; 'display
                   ;; 'antenna
                   ;; 'databases
                   ;; 'samples
                   'modes
                   'ranges
                   'rate
                   'latitude
```

```
                    'longitude
                    'altitude
                    'heading
)
```

Table 13. RsSystem's GUF Slots

Slot Index	Slot Name	Value Type
1	mode display	physical display window
2	mode antenna	physical antenna model
3	mode databases	common terrain databases
4	mode samples	samples per sweep line
5	modes	RsModes
6	ranges	List of RsFloat
7	rate	RsFloat
8	radar's latitude	RsFloat
9	radar's longitude	RsFloat
10	radar's altitude	RsFloat
11	radar's heading	RsFloat

The public interface of the RsSystem class follows:

```
static RsSystem* createSystem(const char* cf, ParserFormFunc func,
            long winId = 0);
static RsSystem* createSystem(const char* cf, ParserFormFunc func,
            const int left, const int right,
            const int bottom, const int top);
RsSystem(void);
virtual ~RsSystem();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void mainLoop(void);
virtual void quit(void);
void redraw(void);
int rate(void) const;
RsWindow* getWindow(void);
const RsWindow* getWindow(void) const;
int getModeIndex(void) const;
void setModeIndex(const int mode);
RsMode* getMode(void);
const RsMode* getMode(void) const;
float getRange(void) const;
int getRanges(float ranges[], const int max) const;
int getRangeIdx(void) const;
void setRangeIdx(int idx);
void printRanges(void) const;
double latitude(void) const;
void latitude(const double lat);
double longitude(void) const;
void longitude(const double lon);
double altitude(void) const;
void altitude(const double alt);
double heading(void) const;
void heading(const double hdg);
double poiLatitude(void) const;
void poiLatitude(const double lat);
```

```
double poiLongitude(void) const;
void poiLongitude(const double lon);
void addInputCallback( int event, RsCallbackProc );
void setAppCallback( RsCallbackProc, RsObject* userData = NULL );
void setGraphicsCallback( RsCallbackProc, RsObject* userData = NULL );
static void redrawCB(RsRadar* sys, RsObject*);
static void quitCB(RsRadar* sys, RsObject*);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* modeChangeMsg;
static const char* shutDownMsg;
static const char* form;
static const RsType type;
```

6.3.8 RsWindow

The RsWindow class manages the physical display window. The RsWindow class is derived from the RsDisplay class. The critical member function of the RsWindow is the initialization, init(). There are three signatures for the init() member function. The init() member function can be called without a window identifier, in which case a window is created for the radar system, and the respective graphical display information is set to defaults. The init() can be called with a window identifier, or with a set of physical screen coordinates, in which all of the graphical display information must be queried and retrieved from the window identifier. The init() member function must be called before any instances of RsDisplay or its derived subclasses are created. Typically, this is handled for the user by the RsSystem::createSystem() member function.

The RsWindow class is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsWindow class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). The slots for the defined form RsWindow consist of the physical width (pixels) and height (lines) of the viewport, the base color index, type of color vector (RGB or HSV), the color mode (colormap or RGB) and the buffer mode (single or double) as is shown within Table 14. The following is the syntax of the defined form for the RsWindow class:

```
(def-form Window
```

```
        'width
        'height
        'colorIndex
        'colors
        'cmode
        'bmode
)
```

Table 14. RsWindow's GUF Slots

Slot Index	Slot Name	Value Type
1	width	physical width of viewport (pixels)
2	height	physical height of viewport (lines)
3	colorIndex	base color index
4	colors	HSV or RGB colors
5	cmode	color mode (colormap or RGB)
6	bmode	buffer mode (single or double)

The public interface for the RsWindow class follows:

```
RsWindow();
virtual ~RsWindow();
static void init(const long wid = 0);
static void init(Screencoord left, Screencoord right, Screencoord bottom,
            Screencoord top);
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.4 Database Library

The Database Library, denoted as libDatabase, provides access classes to the various terrain database file formats for the APG-70 radar simulation. The Database, DED, and Performer libraries use the object-oriented design to exemplify the 'plug and play' features of the source code underlying the APG-70 radar simulation. In this fashion, the RsDatabase provides a common 'outlet' to the radar simulation for any of the 'plugs' which are needed for the specific terrain database files. This design provides maximum flexibility for future growth and reduces the amount of implementation time to a minimum. The DED library, denoted as libDedDbase, provides the 'plug' for the DMA DED terrain database files. The Performer

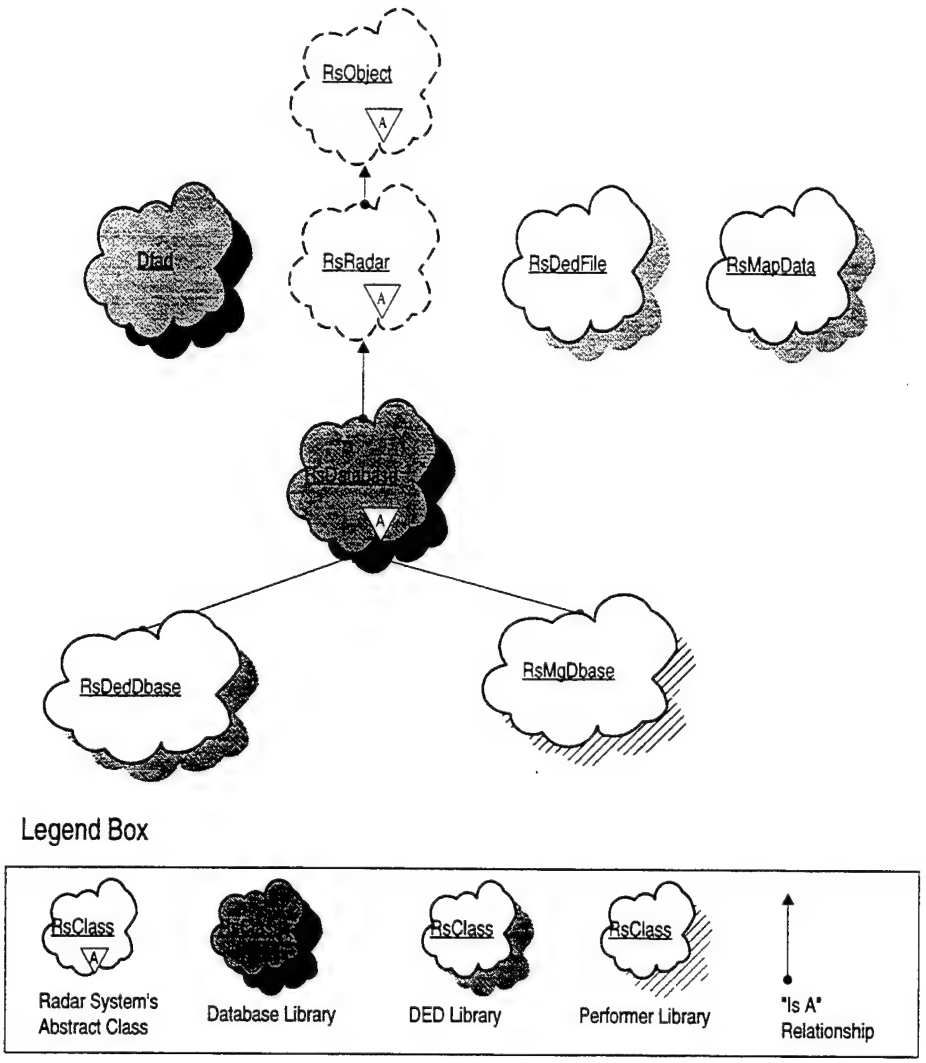


Figure 14. Database, DED, and Performer Library's Class Hierarchies

library, denoted as libPerfDbase, provides the 'plug' for the Multigen and/or Performer polygonal *.flt terrain database files. In order to support another terrain database file format, a derived class of the RsDatabase class would need to be implemented. In order to review the various classes provided within the Database library, see Figure 14. The diagram inidicates which classes are provided by the Database library, the DED library and the Performer library in addition to the inheritance relationships among these classes and libraries.

6.4.1 RsDatabase

The RsDatabase class stands as a class for managing the terrain and culture databases used in the radar simulation. The RsDatabase class derives from the RsRadar class as seen within *Figure 12, Class Hierarchy for RsRadar Derived Subclasses.*

The RsDatabase class is provided so that the interface to the radar simulation is the same regardless of the type of terrain database files. The object-oriented design of the RsDatabase with all of its derived subclasses, RsDedDbase and RsMgDbase, provides maximum flexibility for future growth. Once the interface to the RsDatabase class is understood, the amount of implementation time to provide support for a new terrain database file format is reduced to a minimum. In order to support another terrain database file format other than DMA DED or the Multigen *.flt, a derived class of the RsDatabase class would need to be implemented.

A primary member function of the RsDatabase class is getSweepData() which sets two RsSweep reference pointers as follows: the first one to data from the terrain database, and the second one, to data from the terrain feature database. Therefore, this class provides a point, or hook, whereby terrain feature data can be inserted when needed. The elevation data are in meters and the culture data are DFAD feature ID numbers.

The radar sweep starts at the latitude and longitude values provided and moves along a line in the given angular direction. The length of the sweep is 'range' meters producing a number of sample points, 'np.' These datum inclusive of the center of the gaming area and the center of the database are encapsulated by the RsDatabase class. A variety of member functions are provided to get and set this encapsulated data as follows: getElevation(), latitude(), longitude(), elevation(), getRange(), setRange(), getSamples(), setSamples(), and getCenter(). The getElevation() member function obtains the elevation at a given latitude and longitude of the terrain database. Another member function, makePos() sets an Iris Performer, pfVec3, with x, y, and z in meters relative to the center of the gaming area.

The RsDatabase class is directly supported by the GUF-based language. For the support of GUF, certain functions are provided within the public interface of the RsDatabase class, such as setSlotByIndex(), getSlotByIndex(), formName(), and isFormName(). In addition, the GUF slot table for the RsDatabase follows:

Table 15. RsDatabase's GUF Slots

Slot Index	Slot Name	Value Type
1	latitude	latitude of database center (deg)
2	longitude	longitude of database center (deg)
3	elevation	elevation of database center (ft)
4	samples	number of samples
5	range	range measured in meters (m)

The following is the syntax of the defined form for the the RsDatabase class:

```
( def-form Database
      'latitude
      'longitude
      'elevation
      'samples
      'range
)
```

The public interface for the RsDatabase class follows:

```
enum { maxsweeps = 20 };
enum { invalid_range = -1 };
RsDatabase(void);
virtual ~RsDatabase();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void getSweepData(RsSweep& elev, RsSweep& culture,
            const double lat, const double lon, const float ang);
virtual float getElevation(const double lat, const double lon);
float getRange(void) const;
void setRange(const float rng);
int getSamples(void) const;
void setSamples(const int n);
void getCenter(double* lat, double* lon, float* elev) const;
double latitude(void) const;
double longitude(void) const;
float elevation(void) const;
void makePos(pfVec3 pos, const double lat,
            const double lon, const float alt) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
```

```
static const char* form;
static const RsType type;
```

6.4.2 Dfad

The Dfad class is a base class which does not derive from the RsObject class. The primary purpose of this class is to maintain information regarding the Defense Mapping Agency (DMA) Digital Feature Analysis Data (DFAD). This class is not used by the APG-70 radar simulation in its currently integrated state, since none of the terrain database files used to run the APG-70 RBM PPI or the HRM patch map modes contain any DFAD for the terrain.

The public interface for the Dfad class follows:

```
Dfad(void);
float getReflectivity(const int fid);
const char* getDescription(const int fid);
```

6.5 DED Library

The DED Library of the APG-70 radar simulation contains the support to access the DMA Digital Elevation Data (DED) terrain database files. The RsDedDbase class is a subclass derived from the RsDatabase class. The RsDatabase class is provided so that the interface to the radar simulation is the same regardless of the type of terrain database files. This usage of object-oriented design provides maximum flexibility for future growth and reduces the amount of implementation time to a minimum. The DED library, denoted as libDedDbase, provides the 'plug' for the DMA DED terrain database files which are used for the RBM PPI mode of the APG-70 radar simulation.

6.5.1 RsDedDbase

The RsDedDbase class handles the interface between the DMA DED terrain database files and the APG-70 radar simulation. The RsDedDbase class stands as a derived subclass from the RsDatabase class and as such inherits several of the RsDatabase class features including functionality and data members, such as latitude, longitude, and elevation of the database center, number of samples per sweep, and sweep range. Since the RsDatabase class defines the following member functions as being virtual, update(), setup(), getSweepData(), and getElevation(), the RsDedDbase class has the option to redefine these member functions and

extend the functionality of the RsDatabase class. In actuality, setup(), getSweepData(), and getElevation() are the only redefined member functions inherited from the RsDatabase class.

The defined constant LO_RES_PATH is the path which indicates the default path for the DED terrain database files which are the input into the RBM PPI mode of the APG-70 radar. The path is defined within the RsDedDbase class because a reference to a RsMapData object is privately maintained within this class. As radar sweeps are obtained via the redefined member function get SweepData(), the RsDedDbase class determines whether new DED files need to be loaded using the input position. When new rows or columns of terrain elevation maps need to be loaded, they are created and the current maps are deleted within the RsDedDbase member function loadMaps(). Once the maps are loaded, the terrain elevation posts are obtained via the member function gndscn(). The gndscn() member function determines which DED file is needed to obtain the sweep of elevation posts for the current location, the current range, and the current number of points or samples. The RsDedDbase class currently sets the culture data references to default values.

The RsDedDbase class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are held within the public interface of the RsDedDbase such as setSlotbyIndex(), getSlotbyIndex(), formName(). In addition, the GUF slot table for the RsDedDbase follows exactly the same pattern as the RsDatabase GUF slot table as there are no significant difference within the paramaters for the class when compared to the RsDatabase class (see Table 16).

Table 16. RsDedDbase's GUF Slots

Slot Index	Slot Name	Value Type
1	latitude	latitude of database center
2	longitude	longitude of database center
3	elevation	elevation of database center
4	samples	number of samples
5	range	range measured in meters

Since the RsDedDbase class is inherited from or 'is a' RsDatabase class, the GUF defined form represents this relationship. The GUF defined form syntax follows for the RsDedDbase class:

```
( def-form DedDbase
      is= DataBase
)
```

The public interface for the RsDedDbase class follows:

```
RsDedDbase(void);
~RsDedDbase();
virtual void setup(RsRadar* owner);
virtual void getSweepData(RsSweep& elev, RsSweep& culture,
            const double lat, const double lon, const float ang);
virtual float getElevation(const double lat, const double lon);
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.5.2 RsDedFile

The RsDedFile class manages MultiGen, Inc.'s Digital Elevation Data (DED) file. See FORMAT.DED within the $YARS/database/ded/ directory for the original MultiGen documentation. The RsDedFile class is a base class which does not derive from any other class within the RsObject class hierarchy. The RsDedFile class defines several structures as follows: ss_stdhdr, to contain the standard header file block of 128 bytes, stats, to contain a 32 byte statistics record, and cell_hdr, to contain the 40 byte header for each DMA cell on the disk. The public interface for the RsDedFile class primarily provides access the these data structures and statistics regarding the file of interest.

The public interface for the RsDedFile class follows:

```
ss_stdhdr    stdhdr;          // Standard header
stats        fstat;           // Statistics header
cell_hdr*    cells;           // Array of Cell headers
short**      columns;         // Array of data columns

DedFile(const char *fname);
~DedFile();
int isValid(void) const       { return valid; }
```

```
int rcount(void) const          { return rc; }
void reference(void)            { rc++; }
void unreference(void)          { rc--; }
```

6.5.3 RsMapData

The RsMapData class manages terrain elevation map data. One of the constructors of the RsMapData class loads a number of latitude, 'nlon,' by a number of longitude, 'nlat,' DMA terrain files centered around the input latitude and longitude. The DMA terrain files are given a specific path name so that the terrain files are loaded into DedFile structures. Dlat and dlon are the spacing between files in seconds multiplied by ten. Another constructor provides the ability to load via an existing RsMapData instance. The find_submp() member function returns a pointer to the DedFile structure containing the input latitude and longitude and replies with the spacing between elevation posts in seconds multiplied by ten. The member function chkMapData() is used to check if a new RsMapData needs to be constructed for this input latitude and longitude. The RsMapData class is a base class which does not derive from any other class within the RsObject class hierarchy.

The public interface for the RsMapData class follows:

```
cell_hdr    cell;       // DED cell data
DedFile     **dedFiles; // Array [nx][ny] of DedFile ptr's
RsMapData(  const double lat, const double lon,
            const int dlat,   const int dlong,
            const int nlat,   const int nlong,
            const char* path );
RsMapData(  const RsMapData* p, const double lat,
            const double lon, const char* path);
virtual ~RsMapData();
DedFile* find_submp(const double lat, const double lon,
           double* slat, double* slon) const;
int chkMapData (const double lat, const double lon) const;
```

6.6 Performer Library

The Performer Library of the APG-70 radar simulation contains the support to access the Multigen/Performer polygonal *.flt terrain database files. The Performer Library discussed within the next few paragraphs should not be confused with the Iris Performer Libraries. The RsMgDbase class is a subclass derived from the RsDatabase class. The RsDatabase class is provided so that the interface to the radar simulation is the same regardless of the type of terrain database files. This usage of object-oriented design provides maximum flexibility for

future growth and reduces the amount of implementation time to a minimum. The Performer library, denoted as libPerfDbase, provides the 'plug' for the MutliGen/Performer polygonal terrain database files which are used for the HRM patch map mode of the APG-70 radar simulation.

6.6.1 RsMgDbase

The RsMgDbase class handles the interface between the MultiGen polygonal flight files and the APG-70 radar simulation. The RsMgDbase class derives from the RsDataBase class and therefore inherits several of the RsDatabase class features including functionality and data members, such as latitude, longitude, and elevation of the database center, number of samples per sweep, and sweep range. Since the RsDatabase class defines the following member functions as being virtual, update(), setup(), getSweepData(), and getElevation(), the RsMgDbase class has the option to redefine these member functions and extend the functionality of the RsDatabase class. In actuality, setup(), getSweepData(), and getElevation() are the only redefined member functions inherited from the RsDatabase class.

Table 17. RsMgDbase's GUF Slots

Slot Index	Slot Name	Value Type
1	latitude	latitude of database center
2	longitude	longitude of database center
3	elevation	elevation of database center
4	samples	number of samples
5	range	Range measured in meters
6	files	MultiGen Files
7	path	Path to the MultiGen Files

The RsMgDbase class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are held within the public interface of the RsMgDbase such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsMgDbase class is inherited from or 'is a' RsDatabase class, the GUF defined form and slot table represent this relationship (see Table 17). The GUF defined form syntax follows for the RsMgDbase class:

```
( def-form MgDbase
      is= DataBase
      'files
      'path
)
```

The public interface for the RsMgDbase class follows:

```
RsMgDbase(void);
~RsMgDbase();
virtual void setup(RsRadar* owner);
virtual void getSweepData(RsSweep& elev, RsSweep& culture,
            const double lat, const double lon, const float ang);
virtual float getElevation(const double lat, const double lon);
pfScene* getScene(void);
RsFlatEarth* getEarthModel();
const RsFlatEarth* getEarthModel() const;
pfScene* cull(const pfSphere&);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,
            const int indent = 0, const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

Table 18. RsApg70's GUF Slots

Slot Index	Slot Name	Value Type
1	mode display	physical display window
2	mode antenna	physical antenna model
3	mode databases	common terrain databases
4	mode samples	samples per sweep line
5	modes	RsModes
6	ranges	List of RsFloat
7	rate	RsFloat
8	radar's latitude	RsFloat
9	radar's longitude	RsFloat
10	radar's altitude	RsFloat
11	radar's heading	RsFloat

6.7 APG-70 Library

The APG-70 library, denoted as libApg70, supports the ability to construct a specific APG-70 radar simulation system. The primary support provided by the APG-70 library is the ability to add callback functionality to the radar system. In addition, the programs that initiate the APG-70 radar model simulation exist within this library's directory location. The only class

which exists within the APG-70 library is the RsApg70 class, therefore, a class hierarchy diagram is not needed to show relationships among several classes. Please refer to *Figure 12, Class Hierarchy for RsRadar Derived Subclasses* in order to see where the RsApg70 class is located within the class inheritance of the radar simulation source code.

6.7.1 RsApg70

The RsApg70 class supports the construction of an APG-70 radar system. This class provides the ability to add a callback functionality to the radar system. Primarily adding the capability to accept keyboard input and callback member functions based on specified events. These callback member functions send messages to the respective subsystems within the radar simulation system to manipulate the variable settings of the radar, such as range, scan width, and radar intensity.

The RsApg70 class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are held within the public interface of the RsApg70 such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsApg70 class is inherited from or 'is a' RsSystem class, the GUF defined form and the GUF slot table both represent this relationship (see Table 18). The GUF defined form syntax follows for the RsApg70 class:

```
( def-form Apg70
      is=   System
)
```

The public interface for the RsApg70 class follows:

```
RsApg70(void);
virtual ~RsApg70();
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.8 Off Mode Library

The Off Mode Library, denoted as libOffMode, supports the ability to construct a specific system off mode for the radar simulation. The primary reason for this type of support is to provide the state machine for the radar simulation modes of operation. As a radar system has a set of modes or system states in which it operates, the RsMode class supports the common mode functions and data for the radar simulation. The RsMode class exists within the Radar library, however, the derived subclasses of this mode exist within other libraries, such as the Off Mode library which is detailed within the following paragraphs. The only class which exists within the Off Mode library is the RsOffMode class, therefore, a class hierarchy diagram is not needed to show relationships among several classes. Please refer to *Figure 12, Class Hierarchy for RsRadar Derived Subclasses* in order to see where the RsOffMode class is located within the class inheritance of the radar simulation source code.

6.8.1 RsOffMode

The RsOffMode class stands as a class which is inherited from the RsMode class. The RsOffMode class simulates a generic off mode which truly does nothing within the radar simulation system. The RsOffMode class is provided for upgrade reasons as the current integration status of the RBM PPI and the HRM patch map modes of the APG-70 radar simulation are not tightly coupled.

Table 19. RsOffMode's GUF Slots

Slot Index	Slot Name	Value Type
1	display	RsDisplay
2	antenna	RsAntenna
3	databases	RsDatabase
4	samples	RsInteger

The RsOffMode class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are held within the public interface of the RsOffMode such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsOffMode class is inherited from or 'is a' RsMode class, the GUF

defined form and GUF slot table represent this relationship (see Table 19). The GUF defined form syntax follows for the RsOffMode class:

```
( def-form OffMode
      is= Mode
)
```

The public interface for the RsOffMode class follows:

```
RsOffMode(void);
virtual ~RsOffMode();
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.9 Stand By Mode Library

The Stand By Mode Library, denoted as libStbyMode, supports the ability to construct a specific system stand by mode for the radar simulation. The primary reason for this type of support is to provide the state machine for the radar simulation modes of operation. As a radar system has a set of modes or system states in which it operates, the RsMode class supports the common mode functions and data for the radar simulation. The RsMode class exists within the Radar library, however, the derived subclasses of this mode exist within other libraries, such as the Stand By Mode Library which is detailed within the following paragraphs. The only class which exists within the Stand By Mode Library is the RsStbyMode class, therefore, a class hierarchy diagram is not needed to show relationships among several classes. Please refer to *Figure 12, Class Hierarchy for RsRadar Derived Subclasses* in order to see where the RsStbyMode class is located within the class inheritance of the radar simulation source code.

6.9.1 RsStbyMode

The RsStbyMode class stands as a class which is inherited from the RsMode class. The RsStbyMode class simulates a generic stand by mode which truly does nothing within the radar simulation system. The RsStbyMode class is provided for upgrade reasons as the current integration status of the RBM PPI and the HRM patch map modes of the APG-70

radar simulation are not tightly coupled. It is envisioned that the stand by mode of the radar state machine will facilitate the transition between the RBM PPI and the HRM Patch Map modes when the radar simulation is completely integrated within the IMPACT cockpit.

Table 20. RsStbyMode's GUF Slots

Slot Index	Slot Name	Value Type
1	display	RsDisplay
2	antenna	RsAntenna
3	databases	RsDatabase
4	samples	RsInteger

The RsStbyMode class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are held within the public interface of the RsStbyMode such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsStbyMode class is inherited from or 'is a' RsMode class, the GUF defined form represents this relationship. The GUF defined form syntax follows for the RsStbyMode class:

```
( def-form StbyMode
      is= Mode
)
```

The public interface for the RsStbyMode class follows:

```
RsStbyMode(void);
virtual ~RsStbyMode();
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;9
static const RsType type;
```

6.10 Real Beam Mode Library

The Real Beam Mode Library, denoted as libRealBeamMode, focuses the subsystems defined within the Radar Library needed for theRBM PPI mode of the APG-70 radar simulation. The RsAntenna, RsMode, and RsDisplay subsystems from the Radar Library are extended for the

Real Beam Mode Library into the following classes: RsRbmAntenna, RsRbmDisplay, RsRbmDisplay, and RsRealBeamMode (see Figure 15).

6.10.1 RsRbmAntenna

Since the RsRbmAntenna class stands as a derived subclass from the RsAntenna class, the encapsulated data members, defined within the RsAntenna class, are extended within the RsRbmAntenna class. The RsRbmAntenna class encapsulates a parent radar mode, a state machine, a state table, the size of the state table, a list of scan widths, the maximum scan width index, the current scan width, the scan width of the last iteration, the maximum scan width, a list of scan bars, the maximum scan bar index, and the current scan bar index. The various scan bars provide a small set of scan bar patterns for the simulated radar return data. The current state of the APG-70 RBM PPI mode only uses a single scan bar within its current configuration. As can be seen within the public interface of the RsRbmAntenna class, several member functions provide the ability to 'get' and 'set' the encapsulated data.

In addition to these member functions, the RsRbmAntenna class provides getLeftEdge(), to return the left edge of the maximum scan pattern. The commandedPosition() member function commands the antenna to a computed azimuth and elevation that is based on the scanwidth parameter and the current reference angles provided.

The state machine within the RsRbmAntenna class contains a state table which defines three states, a reset, a sector scan, and a circular scan state. Each of the states have callback member functions, respectively they are named RsRbmAntenna::resetCB(), RsRbmAntenna::sscanCB(), and RsRbmAntenna::cscanCB(). The beforeState() member function runs before any of these other states. In the current configuration of the radar simulation, the sector scan and the reset states are the only ones in use. The circular scan provides for a complete 360° circular scan as identified within the title. Aside from the state callback member functions, the setup() and the update() functions are redefined within the RsRbmAntenna class. The setup() function, called by the parent radar mode, creates an instance of the RsRealBeamMode object. The update() function basically updates a single pass of the state machine.

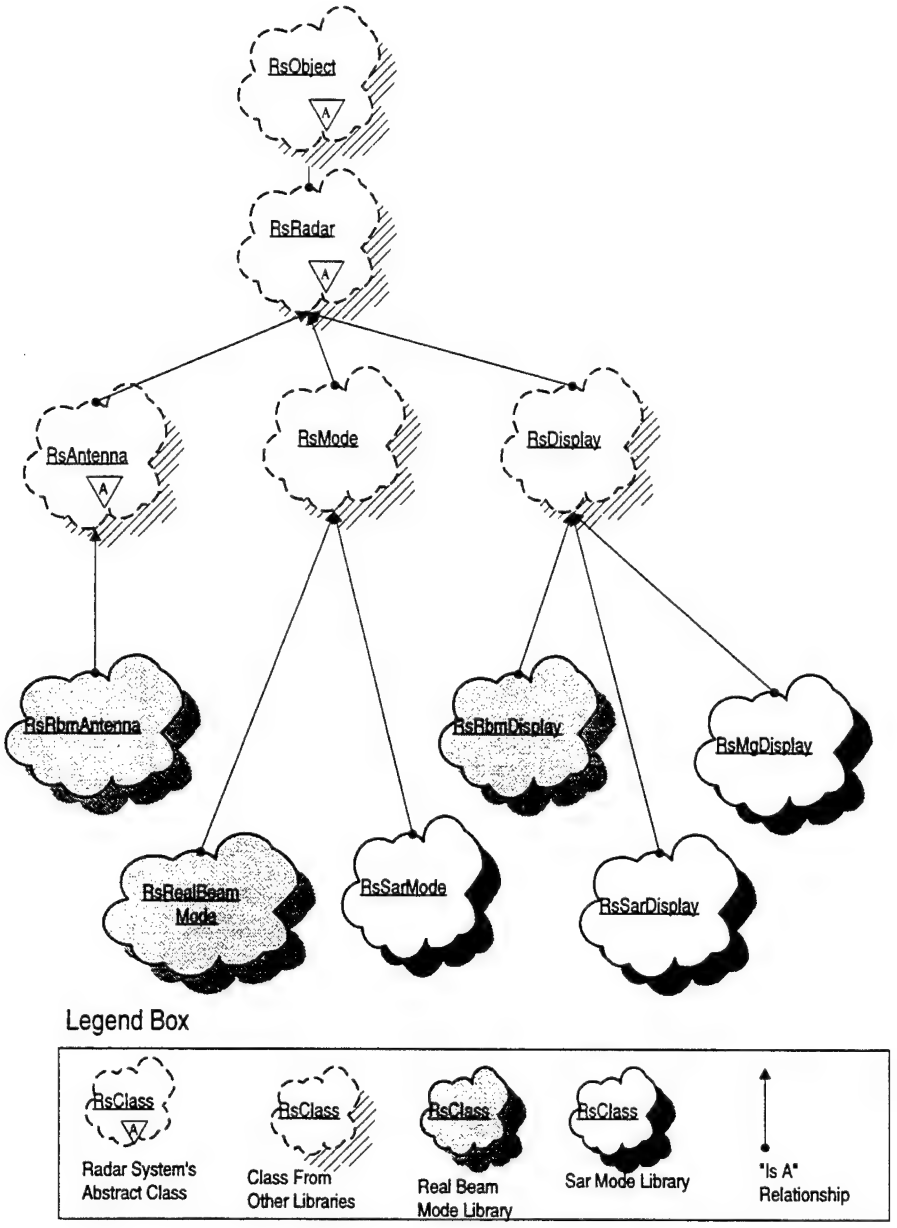


Figure 15. Real Beam and Sar Library's Class Hierarchies

The RsRbmAntenna class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public interface of the RsRbmAntenna class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). In addition, as the RsRbmAntenna class is supported by the GUF syntax, a GUF slot table also exists for instances of the class (see Table 21). The GUF defined form syntax follows for the RsRbmAntenna class:

```
( def-form RbmAntenna
      'scanWidths
      'scanBars
)
```

Table 21. RsRbmAntenna's GUF Slots

Slot Index	Slot Name	Value Type
1	scanWidths	list of floats
2	scanBars	valid scan bar patterns

The public interface for the RsRbmAntenna class follows:

```
RsRbmAntenna(void);
virtual ~RsRbmAntenna();
virtual void setup(RsRadar* owner);
virtual void update(const float dt);
void setScanWidth(const float w);
float getScanWidth(void) const;
int getScanWidths(float widths[], const int max) const;
float getMaxScanWidth(void) const;
float getLeftEdge(void) const;
int getScanWidthIdx(void) const;
int getMaxScanWidthIdx(void) const;
void setScanWidthIdx(int);
int getScanBar(void) const;
int getScanBarIdx(void) const;
void setScanBarIdx(const int idx);
void printRefAngles(void) const;
void printScanWidths(void) const;
void printScanWidth(void) const;
void printScanBar(void) const;
void printScanBars(void) const;
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.10.2 RsRbmDisplay

Since the RsRbmDisplay class stands as a derived subclass from the RsDisplay class, the encapsulated data members, defined within the RsDisplay class, are extended within the RsRbmDisplay class. The RsRbmDisplay class encapsulates range rings for each range, the maximum number of range ring indices, the number of sweep lines per degree, the number of degrees per sweep line, a linked-list of sweeps in order from left to right, the last sweep

added, and a list of free sweeps. Several 'set' and 'get' member functions facilitate the encapsulation of these data members within the RsRbmDisplay class as provided within the public interface. Even though the RsRbmDisplay class provides information regarding range rings, this class does not draw the range rings. The drawing of any of the symbols decorating the radar scan is performed within the wlAPG70OverlayGfx class.

Several member functions provide the support of the actual drawing of the radar return. In order to draw the radar scan, the draw() member functions takes the left most and right sweeps as inputs and determines whether to draw the simulated radar return. If there is no Iris GL window, nothing is drawn. If the angles of both sweeps are the same or if the difference between the two angles is too large, nothing is drawn. Two member functions provide the ability to support single-buffered or double-buffered modes of display output provided that the previously mentioned conditions do not exist.

The update() member function checks the display queue for messages which are relevant to the RsRbmDisplay class. The following messages invoke a change in the display: sweep, erase, rightEdge, and leftEdge.

The simulated radar return scan is stored as a linked-list of radar return sweeps. The head of the linked-list for the radar scan is stored as the left most radar sweep. The tail of the linked-list for the radar scan is stored as the right most radar sweep. Currently, the maximum number of sweeps stands as 800. A new sweep is put in the sweep list based on five conditions. If the sweep list is empty, a new sweep is added to the tail as the first sweep on the linked-list. If the new sweep's angle is less than the left most sweep, a new sweep is put on the head of the list. If the new sweep's angle is greater than the right most sweep, a new sweep is put on the tail of the list. If the new sweep's angle is equal to an existing sweep, the existing sweep is replaced by the new sweep. If the new sweep's angle is between two existing sweeps, the new sweep is inserted between them.

Table 22. RsRbmDisplay's GUF Slots

Slot Index	Slot Name	Value Type
1	width	float
2	height	float
3	rangeRings	list of floats for valid range rings

The RsRbmDisplay class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public interface of the RsRbmDisplay class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsRbmDisplay class is inherited from or 'is a' RsDisplay class, the GUF defined form represents this relationship. The GUF defined form syntax follows for the RsRbmDisplay class:

```
( def-form RbmDisplay
      is= Display
      'rangeRings
)
```

The public interface for the RsRbmDisplay class follows:

```
RsRbmDisplay(void);
virtual ~RsRbmDisplay();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void redraw(void);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.10.3 RsRealBeamMode

The RsRealBeamMode class stands as a class which is inherited from the RsMode class. The RsRealBeamMode class simulates a generic RBM PPI mode of a radar simulation that may not be specific to the APG-70 radar. The RsMode superclass or abstract class provides the

container for an antenna, a terrain database, and a display. Therefore, these subsystems are all inherited from the RsMode class within instances of the RsRealBeamMode class. The RsRealBeamMode class incorporates the following data members privately to handle the simulation of a real beam radar PPI mode: a radar gain factor, a maximum radar return intensity, the number of degrees per radar sweep, the angle of the last radar sweep, the antenna azimuth rate of the last sweep, the left and right most azimuth sweep angles in the current radar scan, and the array of reflectivity values indicated by DFAD ID numbers. As indicated within the public interface of the RsRealBeamMode class, several 'get' and 'set' member functions provide the support to encapsulate these data members.

The setup() member function of the RsRealBeamMode class creates these subsystems if they are not defined via the GUF-based input configuration files. In addition, default values are set for the data encapsulated within the RsRealBeamMode class. The database subsystem is notified of the current configuration parameters, such as range and number of samples. A single radar sweep is processed at initialization for the initial value of the antenna's azimuth.

The primary member function of interest for the RsRealBeamMode class is the update() routine. Initially, the antenna model is updated, the current antenna azimuth angle is obtained and is rounded to the nearest sweep. Sweeps are drawn every getDegreesPerSweep() degrees. When the antenna model has moved into the next sweep line, the database subsystem is notified of the current range and number of samples. Following this notification, a series of radar sweeps are generated via the genSweeps() member function. The genSweeps() member function generates a series of sweep lines from the initial angle given to the second angle given with a specified number of degrees between each radar sweep.

Several member functions support the generation of the radar sweeps within the RsRealBeamMode class. The member function oneSweep() generates a single radar sweep in the direction of the input 'angle.' The member function vbwShadow() computes the effects of the antenna's vertical beam width (VBW) and the effects of terrain shadows. The

vbwShadow() routine uses a sweep of elevation posts measured in meters as input and returns a sweep of zeros at the elevations posts where the terrain is in a shadow or out of the vertical beam. If the terrain is not in a shadow or out of the vertical beam, a sweep of ones is returned at the elevations posts. The angle aspect computer or aac() member function computes the gain effect on the sweep line caused by the angle in which the beam hits the terrain. The aac() member function uses the output sweep from the vbwShadow() member function to filter the number of calculations performed and a sweep of elevation posts for the terrain data at the antenna elevation angle. The member function culture() computes the effect on radar return due to terrain feature data. The culture() member function is provided for future enhancements to the RBM PPI mode of the radar as no feature data is used in the current state of the radar simulation. Lastly, the member function gain() computes the antenna and radar gain effect on the radar sweep return data.

Table 23. RsRealBeamMode's GUF Slots

Slot Index	Slot Name	Value Type
1	display	RsDisplay
2	antenna	RsAntenna
3	databases	RsDatabase
4	samples	RsInteger
5	sweepsPerDegree	number of sweeps per degree

The RsRealBeamMode class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public interface of the RsRealBeamMode class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsRealBeamMode class is inherited from or 'is a' RsMode class, the GUF defined form and GUF slot table represent this relationship (see Table 23). The GUF defined form syntax follows for the RsRealBeamMode class:

```
( def-form RealBeamMode
      is= Mode
      'sweepsPerDegree
)
```

The public interface for the RsRealBeamMode class follows:

```
RsRealBeamMode(void);
```

```
virtual ~RsRealBeamMode();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
float getSweepsPerDegree(void) const;
float getDegreesPerSweep(void) const;
float getSweepAngle(const float angle) const;
// Gain and intensity member functions
float getGainFactor(void) const;
float getMaxIntensity(void) const;
void leftEdge(const float angle);
void rightEdge(const float angle);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout,
            const int indent = 0, const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.11 SAR Mode Library

The SAR Mode Library, denoted as libSarMode, focuses the subsystems defined within the Radar Library needed for the high resolution patch map mode of the APG-70 radar simulation. The RsMode, and RsDisplay subsystems from the Radar Library are extended for the SAR Mode Library into the following classes: RsMgDisplay, RsSarDisplay, and RsSarMode (see Figure 15). As noted throughout the report, the high resolution patch map mode of the F15-E's APG-70 radar system is a synthetic aperture radar (SAR) which means that a large radar antenna pattern is synthesized by taking many snapshots of an area as the radar scan moves along the terrain, using doppler shift information to image process the information, and finally producing an exceptionally high resolution patch map of the terrain.

Table 24. RsMgDisplay's GUF Slots

	Slot Name	Value Type
1	width	float
2	height	float

6.11.1 RsMgDisplay

The RsMgDisplay class was created for intermediate test purposes of the various graphical routines such as the culling, polygon scan conversion, and scan line member functions provided within the Basic library for the HRM patch map output of the APG-70 radar simulation. The RsMgDisplay class uses Iris Performer to provide the scan line, polygon

scan conversion, and culling of the terrain database. As the final implementation of the HRM patch map mode output of the APG-70 radar simulation actually uses the Basic library's graphical routines for speed purposes instead of the Iris Performer library's similar routines, the RsMgDisplay class was created to compare the outputs provided via both libraries.

Even though the RsMgDisplay class is not actually instantiated within the final implementation of the APG-70 radar simulation, the RsMgDisplay class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public interface of the RsMgDisplay class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsMgDisplay class is inherited from or 'is a' RsDisplay class, the GUF defined form and the GUF slot table both represent this relationship (see Table 24). The GUF defined form syntax follows for the RsMgDisplay class:

```
( def-form MgDisplay
      is= Display
)
```

The public interface for the RsMgDisplay class follows:

```
RsMgDisplay(void);
virtual ~RsMgDisplay();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void redraw(void);
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.11.2 RsSarDisplay

Since the RsSarDisplay class stands as a derived subclass from the RsDisplay class, the encapsulated data members, defined within the RsDisplay class, are extended within the RsSarDisplay class. The RsRbmDisplay class encapsulates a linked-list of sweeps in order from left to right, the last sweep added, the number of free sweeps, and a list of free sweeps.

The member function putFreeSweep() and getFreeSweep() facilitate the management of the free sweeps.

Several member functions provide the support of the actual drawing of the radar return. In order to draw the radar scan, the draw() member functions takes the left most and right sweeps as inputs and determines whether to draw the simulated radar return. If there is no Iris GL window, nothing is drawn. If the angles of both sweeps are the same, nothing is drawn. Member functions provide the ability to support single buffered or double buffered modes of display output provided that the previously mentioned conditions do not exist.

The update() member function checks the display queue for messages which are relevant to the RsSarDisplay class. The sweep and erase messages invoke a change in the display since there is not scanning within the HRM patch map mode the leftEdge() and right Edge() member functions are not needed.

The simulated radar return scan is stored as a linked-list of radar return sweeps. The head of the linked-list for the radar scan is stored as the left most radar sweep. The tail of the linked-list for the radar scan is stored as the right most radar sweep. Currently, the maximum number of sweeps stands as 800. A new sweep is put in the sweep list based on the same five conditions as those specified within the RsRbmDisplay class. If the sweep list is empty, a new sweep is added to the tail as the first sweep on the linked-list. If the new sweep's angle is less than the left most sweep, a new sweep is put on the head of the list. If the new sweep's angle is greater than the right most sweep, a new sweep is put on the tail of the list. If the new sweep's angle is equal to an existing sweep, the existing sweep is replaced by the new sweep. If the new sweep's angle is between two existing sweeps, the new sweep is inserted between them.

The RsSarDisplay class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public

interface of the RsSarDisplay class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsSarDisplay class is inherited from or 'is a' RsDisplay class, the GUF defined form and GUF slot table both represent this relationship (see Table 25). The GUF defined form syntax follows for the RsSarDisplay class:

Table 25. RsSarDisplay's GUF Slots

Slot Index	Slot Name	Value Type
1	width	float
2	height	float

```
( def-form SarDisplay
      is= Display
)
```

The public interface for the RsSarDisplay class follows:

```
RsSarDisplay(void);
virtual ~RsSarDisplay();
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void redraw(void);
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
           const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.11.3 RsSarMode

The RsSarMode class stands as a class which is inherited from the RsMode class. The RsSarMode class simulates a HRM patch map mode of a radar simulation or SAR mode. The RsMode superclass, or abstract class, provides the container for an antenna, a terrain database, and a display. Therefore, these subsystems are all inherited from the RsMode class within instances of the RsSarMode class. The RsSarMode class incorporates the following data members privately to handle the simulation of a SAR radar: the number of sweeps, a state machine, a state table, a state table size, a scanline generator, the rotation angle of the SAR patch, a layer stack, a pointer to the layer stack, the SAR patch size in meters, array of SAR patch sizes, the maximum index into the array of SAR patch sizes, the position of interest along the terrain database in world coordinates, the source of the radar beam, location

of the antenna/aircraft, in world coordinates and in scanner/screen coordinates. As indicated within the public interface of the RsSarMode class, several 'get' and 'set' member functions provide the support to encapsulate these data members.

A key difference between the RBM PPI mode and the SAR patch map mode of the APG-70 radar simulation is fact that the SAR uses a state machine instead of the antenna to drive the various sub-modes within the processing of the SAR patch map image. The four states provided with the SAR's local state machine are as follows: a reset state, a database state, a processing state, and an output state.

The setup() member function of the RsRealBeamMode class creates an image array of floats. The number of floats within the image array is the number of sweeps multiplied by the number of sample per sweep. If these values are not defined via the GUF-based input configuration files, the default number of sweeps is 480 and the default number of samples per sweep is also 480. The default patch size is defined at 3200. In addition, an array of possible patch sizes is defined to contain six values of 100, 200, 400, 800, 1600, and 3200.

The primary member function of interest for the RsSarMode class is the update() routine. The update() member function primarily updates the SAR's internal state machine. The state machine internal to the RsSarMode class starts out in the reset state. The reset state basically clears the image array by setting all of the values to 1.0. The state machine then jumps to the database state. Within the database state, the scanner is used to erase the display and then the radar point/position of interest is obtained. The patch size to be used for the SAR image is also obtained. Using the flat earth model, the reference point for the terrain database is obtained. Finally, a bounding sphere is determined with the previously obtained parameters and the Iris Performer math utilities. The original scene is culled using the position of the aircraft as the viewing point and using the point/position of interest on the terrain database as the looking point. The angles and distance between the two points are calculated and then the geometry is placed into the scanner as the complete terrain database is traversed from the parent through the children nodes of terrain information. Finally, the viewing point and the

ranges must be transformed into the same scanner/screen coordinate system such that vector calculations are correct. Once these steps are performed within the database state, the state machine jumps to the processing state. Within the processing state of the SAR patch map state machine, calculations are performed to determine the shadow effects, the aspect angle defined as the angle at which the radar beam hits the object within the terrain, and the color adjustment of the polygon. These bits of information are combined with a noise factor (defaulted to 0.10) and random number generation to add a speckling effect to the image information. Once these calculations are performed, the internal SAR state machine jumps to the output state. Finally, the output state goes through the radar sweeps and passes the display information to the display queues so that the display is updated.

Table 26. RsSarMode's GUF Slots

Slot Index	Slot Name	Value Type
1	display	RsDisplay
2	antenna	RsAntenna
3	databases	RsDatabase
4	samples	RsInteger
5	width	number of sweep lines for SAR image width
5	patchSizes	valid SAR patch sizes by range index

The RsSarMode class is directly supported by the GUF-based syntax within the input configuration files for the APG-70 radar system. In order to provide support for the GUF-based input configuration files, certain member functions are provided within the public interface of the RsSarMode class, such as setSlotbyIndex(), getSlotbyIndex(), formName(), and isformName(). Since the RsSarMode class is inherited from or 'is a' RsMode class, the GUF defined form and the GUF slot table both represent this relationship (see Table 26). The GUF defined form syntax follows for the RsSarMode class:

```
( def-form SarMode
      is= Mode
      'width
      'patchSizes
)
```

The public interface for the RsSarMode class follows:

```
RsSarMode(void);
virtual ~RsSarMode();
int getSweeps(void) const;
```

```
float getPatchSize(void) const;
virtual void update(const float dt);
virtual void setup(RsRadar* owner);
virtual void setSlotByIndex(const int slotindex, const RsObject* obj);
virtual RsObject* getSlotByIndex(const int slotindex) const;
virtual const char* formName(void) const;
virtual int isFormName(const char* name) const;
virtual RsType classType(void) const;
virtual int isClassType(RsType type) const;
virtual void print(ostream& sout, const int indent = 0,
            const int slotsOnly = FALSE);
static const char* form;
static const RsType type;
```

6.12 APG-70 Integration Libraries

The 'wlAPG70' prefix is used for all of the APG-70 radar simulation libraries and respective classes which were developed to perform the integration of the APG-70 stand-alone radar source code into the CSIL's IMPACT cockpit simulation. Only the significant classes developed for the integration are detailed within the following paragraphs. These classes were implemented primarily for the various processes contained within the APG-70 Radar System Software Architecture which is detailed within *Section 5.0, Software Architecture Introduction.*

6.12.1 APG-70 Data Buffer Library

The APG-70 Data Buffer Library contains two classes, the wlAPG70DataBuffer class and the wlAPG70Steerpoint class. The purpose of the APG-70 Data Buffer library is to provide the information for the overlay graphics that decorate the radar scan. In addition the APG-70 Data buffer library's classes provide the information needed from the various system models, such as the navigational and aerodynamic model.

wlAPG70DataBuffer

In order to localize the information for the APG-70 radar simulation, the wlAPG70DataBuffer packages information retrieved from the aerodynamic model, the navigational model, and information from the APG-70 radar state machine. The wlAPG70DataBuffer class basically takes a snapshot of the information required with its update() member function.

The wlAPG70DataBuffer class is based on the data buffer design pattern which has been recently defined within the development of the IMPACT cockpit's software. The intent of the data buffer design pattern is to provide a standard method of creating a specific graphical display without forcing the application code, in this case the radar simulation, to know the details about specific graphical displays such as the radar display. Through the use of the data buffer design pattern a graphical display is only dependent upon the pertinent state information driving that specific format and not dependent upon the particular source of this state information. In this fashion, the application code can specify the source of the data at run time or even incrementally add sources of the data without having to modify the format. An example describing the benefit to the data buffer design pattern usage assumes that a graphical display format uses Scramnet to contain specific values. If the application code needs to be tested, and Scramnet is not currently running, the data buffer design pattern permits the use of shared memory, local memory, or even 'hard-coding' to replace the location of the data store or data store values without having to change the specific graphical display. To provide the data buffer design pattern from a global view, meaning across projects, an additional concrete data buffer class is implemented. The concrete data buffer design pattern provides the ability to defer the source of the data buffer design pattern's information until execution time via a text descriptor. Although the data buffer design pattern, provides these benefits, there is a minor problem with the design. The main problem with the design is that a significant amount of tedious support code needs to be written to support the design. To handle this problem a few scripts have been developed to automate the generation of the support code. In addition, developers must be careful defining the text tags within the namespace so that these text tags remain unique.

The two files used as input to create the access classes that support the data buffer design pattern are APG70RadarDataBuffer.h and APG70RadarEnums.h. These files are stored within the $YARS/access directory. The a list of the files which were automatically created to support the wlAPG70DataBuffer class follows:

```
APG70RadarDataAccess.c++
APG70RadarDataAccess.h++
APG70RadarDataWrapper.c++
```

```
APG70RadarDataF77Wrapper.c++
APG70RadarDataInterface.c++
APG70RadarDataInterface.h++
```

In addition to the use of the data buffer design pattern, the Rogue Wave vector template, RWTValVector, class is used within the wlAPG70DataBuffer class. The information which is encapsulated with the wlAPG70DataBuffer class includes the following: an array of navigational steerpoints, the number of steerpoints, the ownship's latitude, the ownship's longitude, the ownship's barometric altitude, the ownship's true heading, the reference latitude, the reference longitude, the radar range, the antenna elevation angle, the antenna azimuth angle, the radar scan width, the radar mode, declutter status, and the current steering point. As can be seen by the public interface of the wlAPG70DataBuffer class, several 'get' functions are provided.

The public interface for the wlAPG70DataBuffer class follows:

```
wlAPG70DataBuffer(void);
virtual ~wlAPG70DataBuffer();
virtual void Update(void);
wlAPG70Steerpoint GetSteerpoint (int ident) const;
int GetNumberOfSteerpoints (void) const;
double GetOwnshipLatitude (void) const;
double GetOwnshipLongitude (void) const;
float GetOwnshipBaroAltitude (void) const;
float GetOwnshipTrueHeading (void) const;
double GetRefLatitude (void) const;
double GetRefLongitude (void) const;
float GetRadarRange(void) const;
float GetAntennaElevAngle(void) const;
float GetAntennaAzimuthAngle(void) const;
int GetScanWidth(void) const;
APG70RadarEnums::RadarModes GetRadarMode(void) const;
int GetDeclutterOn(void) const;
int GetCurrentSP(void) const;
```

wlAPG70Steerpoint

The wlAPG70Steerpoint class is primarily a support class used within the wlAPG70DataBuffer class to encapsulate the steerpoints also known as waypoints which are obtained via the navigational model. The wlAPG70Steerpoint class provides the ability to get the latitude, longitude, identifier, and type information associated with each instance of the class. The identifier is a character string of eight characters which labels the waypoint.

The identifiers are typically characters strings containing numbers, such as '1.0,' '2.0,' or '2.1.' The wlNavEnums class specifies the waypoint.type.

The public interface for the wlAPG70Steerpoint class follows:

```
wlAPG70Steerpoint(void);
~wlAPG70Steerpoint(void);
wlAPG70Steerpoint(const wlAPG70Steerpoint&);
wlAPG70Steerpoint& operator=(const wlAPG70Steerpoint&);
double GetLatitude(void) const;
double GetLongitude(void) const;
const char* GetIdent(void) const;
wlNavEnums::WayPointType GetType(void) const;
void SetLatitude(const double);
void SetLongitude(const double);
void SetIdent(const char*);
void SetType(const wlNavEnums::WayPointType);
```

6.12.2 APG-70 Format VARS Library

The APG-70 Format VARS Library contains only a single class, the wlAPG70FormatVars class. The primary reasons for the wlAPG70FormatVars class are to instantiate the state variables, the APG-70 data buffer class, and the waypoint class and provide an interface to manipulate the state variables. The state variables maintain the information displayed within the window sets for the APG-70 radar display format. These window sets are drawn via the cockpit display manager with the information contained within these variables. The state variables may exist in the form of enumerated values, integers, and floating point numbers. The window sets for the APG-70 radar display format contain the current text information regarding the state of the radar simulation, such as selected radar range, radar mode, radar declutter status, radar gain, radar display brightness, radar scan, radar frequency band and channel. Specific minimum and maximum constants and the enumeration for the bezel switches specific to the real beam mode are defined within the public interface such that other classes have access to these values.

The public interface for the wlAPG70FormatVars class follows:

```
wlAPG70FormatVars(void);
virtual ~wlAPG70FormatVars();
void Init(void);
void Update(void);
void ToggleDeclutter(void);
void CycleGain(void);
void CycleBright(int up_flag);
```

```
void CycleMode(void);
void CycleCurs(void);
void CycleScan(void);
void CycleIPVU(void);
void ToggleVisibility(void);
void VTR(void);
void CycleRange(int up_flag);
void SPI(void);
void ToggleSniff(void);
void CycleFrequencyBand(void);
void CycleFrequency(void);
void PrintType(const char *,int);
void SetRange(const APG70RadarEnums::Ranges range);
void SetRadarMode(const APG70RadarEnums::RadarModes mode);
void SetScanWidth(const APG70RadarEnums::ScanWidths scanwidth);
void SetDeclutterOn(const int flag);
void SetIPVUMode(const APG70RadarEnums::IPVUModes pvumode);
void SetRadarGain(const int igain);
void SetFreqBand(const int freqBand);
void SetFreqChannel(const APG70RadarEnums::FreqChannel freqChannel);
void SetCursorFunction(const APG70RadarEnums::CursFunctions func);
void SetDisplayBrightness(const int brightness);
APG70RadarEnums::Ranges LimitRange(const APG70RadarEnums::Ranges range);
APG70RadarEnums::ScanWidths LimitScanWidths(const
APG70RadarEnums::ScanWidths scanwidth);
APG70RadarEnums::RadarModes LimitRadarMode(const
APG70RadarEnums::RadarModes mode);
APG70RadarEnums::IPVUModes LimitIPVUMode(const APG70RadarEnums::IPVUModes
imode);
int LimitRadarGain(const int igain);
APG70RadarEnums::FreqChannel LimitFreqChannel(const
APG70RadarEnums::FreqChannel freq);
APG70RadarEnums::CursFunctions LimitCursorFunction
            (const APG70RadarEnums::CursFunctions curs);
int LmitFreqBand(const int band);
int LimitToggle(const int flag);
int imitDisplayBrightness(const int brightness);
const int MAX_GAIN = 3;
const int MIN_GAIN = 0;
const int MAX_FREQ_BAND = 8;
const int MIN_FREQ_BAND = 1;
const int MAX_BRIGHTNESS = 15;
const int MID_BRIGHTNESS = 8;
const int MIN_BRIGHTNESS = 0;
enum rbm_names  {
            ONE_BTN_RBM,      // non-functional in rbm
            DECLUTTER_BTN,    // declutter toggle
            GAIN_BTN,         // select gain 3-0
            HI_BRIGHT_BTN,    // increase brightness 0-15
            LO_BRIGHT_BTN,    // decrease brightness 0-15
            MODE_BTN,         // mode select: RBM,HRM,PVU,BCN
            CURS_BTN,         // cursor function select:
                              // MAP,UPD8,TARGET,CUE
            MAP_BTN,          // non-functional in RBM, however,
                              // size of display window selected is
                              // placed above PB 8
            SCAN_BTN,         // FULL(100), HALF(50), QTR(25)
            TEN_BTN_RBM,      // non-functional in rbm
            MENU_BTN,         // Return to Main Menu
            VTR_BTN,          // Video Tape recording toggle
            HI_RANGE_BTN,     // increase range:
```

```
                        // 4.7, 10, 20, 40, 80, 160
LO_RANGE_BTN,           // decrease range:
                        // 4.7, 10, 20, 40, 80, 160
FIFTEEN_BTN_RBM,        // non-functional in rbm
IPVU_BTN,               // toggle IPVU
SPI_BTN,                // sequence point selection
SNIF_BTN,               // toggle SNIFF
FREQ_BAND_BTN,          // frequency band select A-E
FREQ_BTN                // frequency scheme select 1-8
};
```

6.12.3 APG-70 Input Monitor Library

The APG-70 Input Monitor Library contains a single class, the wlAPG70InputMonitor class. The APG-70 Input Monitor Library provides an Iris GL window ability to queue inputs from the keyboard. The keyboard inputs accepted by the input monitor are mapped directly to the bezel switches which surround the APG-70 radar display in the F-15E aircraft. In addition, the escape key, 'ESC,' and the window exit event are used to exit the input monitor program.

wlAPG70InputMonitor

The wlAPG70InputMonitor class handles requests which are made by keyboard inputs. Shared Memory is used to store the information regarding inputs from the keyboard. When wlAPG70InputMonitors are instantiated, an instance can initialize the shared memory or attach to it. If the instance is initializing the shared memory, an Iris GL window is opened to monitor keyboard input. The main_loop() member function continuously runs while the running flag is TRUE.

The check_input() member function monitors for keyboard input of the defined keys or the window quit event. When a recognized key button press event is detected, the shared memory data store is updated. The instance also maintains a local copy of the shared memory, for comparison purposes. Member functions are provided which update the shared memory when inputs are made. The member function same_mem() is provided so that the user can compare the two memory stores and determine if inputs were made. In addition, member functions are also provided which update the shared memory when inputs are made. Shared memory is used to store the input monitor values for process communication purposes. The APG-70 radar state machine process uses the information provided within this shared memory initialized via the APG-70 input monitor process. The APG-70 input

monitor process is simply a program which instantiates an object of the wlAPG70InputMonitor class and uses the constructor which initializes the shared memory area at the address 0x74000.

The public interface for the wlAPG70InputMonitor class follows:

```
wlAPG70InputMonitor(int msize=20, int attach_flag=0);
~wlAPG70InputMonitor(void);
void main_loop(void);
void output_vars(void);
void copy_upd8(void);
int* ret_hits(void);
int cnt_hits(void);
int same_mem(void);
```

6.12.4 APG-70 Overlay Graphics Library

The APG-70 Overlay Graphics Library primarily contains a single class, the wlAPG70OverlayGfx class. The overlay graphics are instantiated by the wlAPG70Radar class which was automatically generated via scripts to support the format design pattern used by all the formats within the IMPACT cockpit simulation. The wlAPG70Radar class is publicly derived from the wlFormat class. In addition to the wlAPG70Radar class, the wlAPG70RadarGraphics class was automatically generated via scripts in order to comply with the way the formats are currently implemented within the IMPACT cockpit simulation. The wlAPG70RadarGraphics class instantiates the wlAPG70OverlayGfx object in addition to a RsDisplaySys object. The RsDisplaySys object is used to display the radar scan produced with the radar simulation model while the wlAPG70OverlayGfx object is used to display the graphical symbols which decorate the radar scan.

wlAPG70OverlayGfx

The primary member function within the wlAPG70OverlayGfx class is DrawMe(). The DrawMe() member function calls all of the other member functions which decorate the radar scans with graphical objects, such as the radar range arcs, the zero azimuth line, the antenna elevation scale, the antenna azimuth scale, the sequence point symbols, the steerpoint symbols, the target point symbols, and the aim point symbols. These graphical symbols use the Iris GL in order to draw these symbols on the radar display format.

The public interface for the wlAPG70OverlayGfx class follows:

```
wlAPG70OverlayGfx(const wlColorMap& aMap);
virtual ~wlAPG70OverlayGfx();
void ScaleForMMDrawing(void);
void MMOrtho(void);
void WorldOrtho(void);
void DrawMe(void);
void DrawRangeArcs(void);
void DrawTic(float x, float y, float size);
void DrawTicCenter(float x, float y, float size);
void DrawZeroAzimuthLine(void);
void DrawAntennaElevationScale(float x, float y, float size);
void DrawAntennaElevationCaret(float x, float y);
void DrawAntennaAzimuthCaret(float x, float y);
void DrawAntennaAzimuthLimits(float x, float y);
void DrawSequencePoints(void);
void DrawSteerPoint(void);
void DrawTargetPoint(void);
void DrawAimPoint(void);
int IsCurrentSP(const int sp) const ;
```

6.12.5 APG-70 State Machine Library

The APG-70 State Machine Library only contains a single class, the wlAPG70StateMachine class. A program which instantiates an object within the wlAPG70StateMachine class also exists within the same directory area. In its current state, the APG-70 state machine process continues to run until it is killed or stopped by the operating system.

wlAPG70StateMachine

The wlAPG70StateMachine class is publicly derived from the RsStMach class that is defined within the Basic Library. The RsStMach class provides the ability to simulate a state machine through the use of a state table. The wlAPG70StateMachine class facilitates the various modes of the APG-70 radar simulation. The wlAPG70StateMachine class attaches to shared memory in order to determine the inputs from the APG-70 input monitor process. The wlAPG70StateMachine class initializes the wlAPG70FormatVars object and sets the appropriate values for the RBM PPI mode of the APG-70 radar simulation into the various parts of the wlAPG70FormatVars object. The input is checked within the BeforeStateFunc() member function. Once inputs are detected, the format state variables are updated within the instantiated wlAPG70FormatVars object. Five states and respective callback member functions are provided with the current implementation of the APG-70 radar state machine as follows: reset, rbm, hrm, bcn, and pvu. Although these states are provided within the

wlAPG70StateMachine class, they are not tightly coupled within the current implementation. The AfterStateFunc() member function that is executed after the different states are entered updates the format state variables from the Scramnet area.

The public interface for the wlAPG70StateMachine class follows:

```
wlAPG70StateMachine(void);
~wlAPG70StateMachine(void);
void output_StateMachine(void);
int isRunning(void) const;
void QuitRunning(void);
```

7. RESULTS AND RECOMMENDATIONS

The results of the work accomplished under Delivery Order (DO) 0013 of the Pilot Factors Contract F33615-93-D-3800 proved successful. The following information provides the status of work and recommendations for future development of the APG-70 radar simulation.

7.1 Status of Software Development

The experimental development of the APG-70 radar simulation, specifically the high resolution mapping (HRM) patch map mode otherwise known as a synthetic aperture radar (SAR), resulted in the accomplishment of six significant milestones. To effectively indicate the overall success of the project, the milestones are explained and their respective status revealed.

Milestone 1—Captured APG-70 Radar PVI Description

Significant effort of the project involved the compilation of various reference documents regarding the pilot-vehicle interface (PVI) for the APG-70 radar system, as included in *Appendix A, Air-To-Ground Radar Pilot-Vehicle Interface.* This information must be enhanced and compiled into a control and display design document to indicate the IMPACT team's exact needs for the level of integration required to further develop the current APG-70 radar simulation.

Milestone 2—Benchmarked Software Technologies for RBM PPI

Efforts involved the benchmarking of products that were either available off-the-shelf or via the Internet. Object-oriented design was chosen because of the flexibility and reusability offered through the technology. To provide the flexibility of the radar system, it was determined that configuration files would provide the communication method to the APG-70 radar simulation. The native Iris Graphics library and the native Iris Performer's math library and their respective data structures were selected because of their optimized speed for the Silicon Graphics hardware platform. Experiments for texture memory were extensive and

determined that texture memory was not fast enough for the RBM of the radar simulation. Once the various technologies were evaluated, software implementation was initiated.

Milestone 3—Developed and Integrated the RBM PPI Map into the IMPACT Simulation

The real beam map (RBM) plan position indicator (PPI) mode of the APG-70 radar was successfully implemented in software. The stand-alone portion of the RBM does not include the overlaid symbology. However, the integration of this version of the software into the IMPACT simulation added the display symbology such as the radar range rings, the zero azimuth line, and the sequence points. In addition to the display symbology, the integration into CSIL's IMPACT simulation provided keyboard control of the various parameters of the RBM PPI. An input monitoring program monitors keyboard input to control parameters, such as antenna elevation, range, scan width, mode, cursor function, declutter toggle, display brightness, channel frequency, and sniff mode. The ability to drive the APG-70 radar simulation with the dynamic position of the aircraft, as obtained via the access class of the aerodynamic model, was also integrated into the IMPACT simulation. The digital elevation data files are loaded in memory for the gross terrain features of the RBM PPI mode of the APG-70 radar simulation so that the terrain database driving the radar simulation corresponds with the database driving the visual scene. The terrain digital elevation data files used for the RBM PPI are some of the same files used to create the database that drives the visual scene. When the initial version of the RBM was delivered, it was based on the first version of the configuration files. Therefore, the RBM PPI mode currently integrated is based on the message-based input configuration files.

Milestone 4—Reviewed Technologies for RBM and SAR

Attendance at the Association for Computing Machinery (ACM) Special Interest Group for Graphics (SIGGRAPH) '95 Conference and the 17th Interservice/Industry Training Systems and Education Conference (I/ITSEC) created another resource for obtaining state-of-the-art graphics and simulation knowledge bases. As a result, several SAR-related and graphics-related algorithms and papers were reviewed in preparation for the implementation of the RBM and the HRM portion of the APG-70 radar simulation. The primary outcome of

these knowledge base resources were quadfold by introducing the Grand Unified File (GUF) Format, the virtual synthetic aperture radar (VSAR) technique, wavelets, and the Open Graphics Library (GL). For more information concerning these technologies, refer to *Section 4, Software Design Approach.*

Milestone 5— Developed and Integrated the Stand-Alone HRM Patch Map into CSIL

Additional software was developed to perform initial visualization of the polygonal terrain database files. The VSAR algorithm indicates that the visual orthographic view is needed before the SAR mathematics can be performed for the shadowing and special effects provided by the SAR. However, the final implementation of the SAR or the HRM patch map of the APG-70 radar only performs one pass of the terrain database. The visualization of the orthographic view into memory was performed by implementing scan line, clipping, and culling algorithms for the terrain database. Within the first pass of the terrain database, the necessary mathematics were performed and only one display output window is needed in the final implementation. As common functionality between the RBM and the HRM radar source was realized, source code from the initial version of the real beam code was modified, involving the development of more abstract classes that are common to the RBM and the HRM modes. The classes are contained in the Radar Library. Also, since the GUF-based language is used for the second version of the radar system, the format of the configuration of the input files was also modified.

Currently, the HRM patch map mode of the radar has not been completely integrated into the IMPACT simulation. This HRM of the delivered system has been configured with GUF-based input files so that the generated image uses the terrain database that drives the IMPACT simulation and therefore correlate to the visual scene image. The MultiGen loader is used to store the database flight files in memory for the HRM patch map mode of the APG-70 radar simulation. The HRM patch map mode uses the same MultiGen flight file created under the IMPACT project, which drives the visual scene for the IMPACT simulation, and the RBM PPI mode uses the corresponding IMPACT digital elevation data files. However, just like the stand-alone version of the RBM, the stand-alone version of the HRM patch map mode does not have the respective overlaying symbology. This symbology

and the ability to use a cursor control via hands-on-throttle-and-stick (HOTAS) to command the HRM patch map mode will be provided when the APG-70 radar simulation is integrated completely into the IMPACT simulation. The newly delivered version of the GUF-based real beam mode is not completely integrated into the IMPACT simulation. However, GUF-based configuration files have been created for the RBM and therefore the second stand-alone version of the RBM has been tested and works within the IMPACT simulation.

Milestone 6— Documented Source Code

A significant effort involved the documenting of the source code for the stand-alone primary radar model. As the radar simulation model was developed by Hughes-Training Inc., Veda Inc., learned, integrated, and documented the two versions of the subcontracted software. In addition to the documentation, Hyper-Text Markup Language (HTML) files were created so that the source code would be browsable in a world wide web (WWW) page format via Netscape. In addition, the de facto method of documenting object-oriented software is with Grady Booch's notation; therefore, the software entitled Visio 4.0 and its add-on package, Advanced Software Diagrams, were purchased so that Booch's notation was electronically captured within the documentation of the source code.

7.2 Recommendations

Although there were many successes of the software implementation of the APG-70 radar simulation, enhancements to the current status of the simulation will provide a more realistic version of the APG-70 radar of the F-15E thereby providing human factors engineers with the necessary information to accurately determine pilot workload. The following recommendations are divided into two categories: near-term and long-term.

7.2.1 Near-Term

Near term recommendations are considered necessary to achieve a completely interactive and realistic APG-70 radar simulation.

Precisely Define Software Requirement Specification for the APG-70 Radar Simulation

Extensive efforts remain to precisely indicate the specific functions and symbology that are required within the simulation of the APG-70 radar. *Appendix A, Air-To-Ground Radar Pilot-Vehicle Interface* provides an overview of the APG-70 radar PVI within the F-15E and initiates a collaboration of reference materials.

Implement the Entire HRM Mode

The HRM patch map mode should be completely integrated into the IMPACT cockpit. The symbology for this mode should be displayed and the ability to command a high resolution patch map from the RBM should be integrated into the current software. The following information provides recommendations for achieving a completely interactive radar simulation within the cockpit:

Create a virtual cursor that can be passed among the various display formats within the simulation. To actualize this capability, the analog values provided by the hardware, which drive the cursor, will need to be placed into the data union class.

As human factors engineers and operational specialists determine requirements for the complete integration and actual mechanization of the bezel switches for the radar into the IMPACT cockpit, the specific hardware technology, touch screen or otherwise, must be integrated. In addition to the cursor interaction analog signals obtained from the HOTAS, the surrounding bezel switches of the radar display will need to be placed into a data union so that the touch screen signals provide the interaction to the radar that is currently simulated via the keyboard and the APG-70 input monitoring process. The cursor functions of the RBM and the HRM modes are not addressed within the current version of the simulation. These functions would add significant realism and pilot-in-the-loop interaction to the radar simulation.

Several features of the HRM mode of the APG-70 radar system are not currently addressed such as the ability to store more than a single patch map. From a pilot's perspective within the cockpit, the current state of the HRM patch map mode is not interactive. Therefore, the

user interaction with the HRM PPI and patch map modes need to be implemented in software as the HRM patch map mode currently runs in a stand-alone mode.

The update rate is another concern for the complete integration of the HRM into the IMPACT cockpit simulation. All of the formats within the simulation are currently rendered in double-buffered mode where the current stand-alone version of the HRM patch map is rendered in single-buffered mode. Therefore, once the current version of the HRM patch map is incorporated into the IMPACT simulation, the update rate may be too slow.

Identify SAR Material Reflectivity Codes

Since the HRM patch map mode of the radar simulation was initially developed and optimized for a monochrome/grayscale color-based terrain database, when the SAR image is generated for the IMPACT terrain database, the output is not optimal because the IMPACT terrain database is a full color database, which does not maintain the reflectivity of the materials within the database. To best optimize the output of the HRM patch map of the radar simulation, the database needs to be enhanced to store the material reflectivity information for the various objects within the database. Once this information is stored within the terrain database, the HRM simulation will have to be modified to capture and enhance the SAR output respectively to the material reflectivity codes stored within the database. The next version of MultiGen, MultiGen II, and Iris Performer 2.0 will provide the capability to store this information within the terrain database. Iris Performer 2.0 is a major release with several new features. One new feature is the C++ application program interface that will extend the advantages of object-oriented programming paradigm, which is used throughout the radar system, from the math library to the vector data types. Following the upgrade paths of MultiGen and Iris Performer would only enhance the performance of the IMPACT cockpit simulation and provide more advanced capability. Therefore, another recommendation would include the addition of the material reflectivity information into the IMPACT terrain database along within the upgrade to MultiGen II and Iris Performer 2.0. However, it should be noted that the upgrade to the recently released MultiGen II and Iris Performer 2.0 would affect the entire IMPACT cockpit simulation because the visual scene is also dependent on these tools. There may be significant changes to the current version of the

IMPACT database along with more modifications to the software within the IMPACT cockpit simulation, which drives the out-the-window visual scene.

7.2.2 Long-Term

Long-term recommendations would optimize and increase the fidelity of the APG-70 radar simulation computer resources, and in addition to providing the ability to handle moving objects within the database, provide hardware platform independence.

Open GL for Hardware Platform Independence

The software source code for the RBM PPI and HRM patch map modes of the APG-70 radar simulation are currently owned by the United States Air Force (USAF). As the USAF has so many facilities that may need the capability provided by the software source code, it may prove beneficial to choose a path where computer platform independence is realized. At SIGGRAPH '95 a course was attended, promoting this path of hardware platform independence via the Open Graphics Library (Open GL) created by Silicon Graphics. Since this library is platform independent, it is not optimized for any native computer hardware and may prove to be slower in terms of update rate. However, as Silicon Graphics is the creator of Open GL and bases the library on their own Iris GL, the difference in speed may not be significant or even detectable by the human eyes. Open GL would need to be benchmarked against Iris GL, and once the decision was made to go with Open GL, a significant portion of the IMPACT cockpit simulation would also need to be modified.

Develop Texture Memory for HRM

Since the HRM mode of the APG-70 radar simulation is not completely integrated into the IMPACT cockpit simulation, the concern for the update rate is not a proven issue. However, since the stand-alone portion of the HRM patch map is being drawn in a single-buffered mode where the current display formats within the cockpit simulation are all drawn in a double-buffered mode, the update rate may become an issue when the SAR portion of the APG-70 radar is integrated into the complete simulation. One way to solve this potential problem is to use the texture memory available for the HRM. In addition, texture memory

could be used to provide the storing functionality for the HRM patch maps which is not currently implemented.

Consider Wavelet Technology for RBM Optimization

Another long-term goal is to optimize the method in which data is stored for the RBM mode of the APG-70 radar simulation. Wavelet technology is a highly effective method of storing data. If the wavelet technology were implemented, the information for the terrain database would be stored in an efficient method and perhaps the simulation of the RBM would also be able to handle ranges larger than 160 nautical miles. Wavelet Technology is addressed in more detail in *Section 3, Software Design Considerations.*

8. CONCLUSIONS

In concluding this report, it is anticipated that the audience would have a better appreciation of the variety of tasks which were involved in the development of the APG-70 radar simulation. One of the key objectives of the report was to provide software intensive information concerning the APG-70 radar simulation so that a proficient software engineer would be able to thoroughly understand the software design and the implementation without having to review the source code. However, as a supplement to this report HTML files are provided on-line on the CSIL file server so that the source code can easily be viewed if necessary.

Other objectives of this final report included the revelation of the current status of the APG-70 radar simulation in addition to specifying the near-term and long-term recommendations. It is strongly suggested that the recommendations be pursued so that the APG-70 radar simulation becomes a truly integrated and significant enhancement to the IMPACT cockpit simulation. It is also recognized that the long-term recommendations will provide overall enhancements within the CSIL facility.

9. REFERENCES

ACM SIGGRAPH. Annual Conference Series: Course Notes CD-ROM. August 6-11, 1995. Los Angeles, CA.

Air Combat Command. Academic Student Workbook: Air-To-Ground Radar, AGR-2E, APG-70 A/G Introduction, (OPR: Det 1, 4444 Ops Sq/F-15 Division). April 1992. Luke AFB, AZ.

Air Combat Command. Academic Student Workbook: Air-To-Ground Radar, AGR-1E, A/G Radar Basics, (OPR: Det 1, 4444 Ops Sq/F-15 Division). June 1992 Luke AFB, AZ.

Air Combat Command. Academic Student Workbook: Air-To-Ground Radar, AGR-3E, Modes, Controls, and Displays, (OPR: Det 1, 4444 Ops Sq/F-15 Division). July 1993. Luke AFB, AZ.

Booch, G. Object-Oriented Analysis and Design With Applications, (Library of Congress Catalog Card No.: 80-24311/ISBN: 0-201-14468-9). 1994. The Benjamin/Cummings Publishing Company, Inc. Redwood City, CA.

Fischler, S., Helman, J., Jones, M., Rohlf J., Schaffer, A., Tanner, C. IRIS Performer Reference Pages, (Document No.: 007-1681-020). 1994. Silicon Graphics, Inc. Mountainview, CA.

Foley, J.D. and Van Dam, A. The Systems Programming Series: Fundamentals of Interactive Computer Graphics, (ISBN: 0-201-14468-9). 1982 (Reprinted March 1983). Addison-Wesley Publishing Company, Inc.

Glassner, A. S. Graphics Gems, (ISBN: 0-12-286165-5). 1990. Academic Press, Inc. Palo Alto, CA.

Hartman, J. and Creek, P. IRIS Performer Programming Guide, (Document No.: 007-1680-020) . 1994. Silicon Graphics, Inc. Mountainview, CA.

McDonnell Aircraft Company. F-15E Human Engineering Design Approach Document - Operator, (Document No.: MDC A9606). December 11, 1985 (Revision Date: May 1, 1988). St. Louis, MO.

McDonnell Douglas Aerospace. USAF Series F-15E Aircraft: Non-nuclear Weapon Delivery Manual, (Technical Order: TO IF-15E-34-11). September 15, 1991 (Revision Date: April 15, 1993). Robins AFB, GA.

Montecalvo, A.J., Redden, M.C., Rolek, E.P. Orr, H.A., Barbato, G. Integrated Mission Precision Attack Cockpit Technology (Impact) Phase I: Identifying Technologies For Air-To-Ground Fighter Integration, (Document No.: WL-TR-94-3143). October 1994. Wright-Patterson AFB, OH.

The American Defense Preparedness Association and The National Security Industrial Association. 17th Interservice/Industry Training Systems and Education Conference Proceedings and Exhibits. November 13-16, 1995. Albuquerque, New Mexico.

Veda Incorporated. Integrated Mission Precision Attack Cockpit Technology (Impact) Mission Analysis and Interface Requirements Definition Report, (Document No.: 63654-94U/P61200). June 30, 1994. Dayton, OH.

APPENDIX A

AIR-TO-GROUND RADAR
PILOT-VEHICLE INTERFACE

AIR-TO-GROUND RADAR PILOT-VEHICLE INTERFACE

Table 1 highlights the performance and purpose of the RBM and the HRM modes of the F15E's APG-70 radar system. The two modes that were incorporated into the Wright Laboratory CSIL facility are the RBM PPI and the HRM patch map. To provide the basic SAR simulation capability in regards to mathematical modeling for the visual display, the HRM patch map mode was developed. This section describes the basic radar format and the common symbology between the RBM and HRM modes as reference data for future development of the APG-70 radar simulation. The commonality between the RBM and HRM modes are primarily within the PPI format. Following the common description, both modes are detailed.

Table 1. APG-70 Primary Radar Modes

Mode	Performance		Purpose
Real Beam Map	**Resolution:** **Range:** **Azimuth:** **Maximum Range:**	127 FT 4.7 NM 2.5 deg 160 NM	• Gross Terrain Features for Navigation • Weather Detection • HRM Cueing
High Resolution Map	**Resolution:** to 20 NM **Resolution:** to 160 NM	8.5 FT (R/AZ) 127 FT (R/AZ)	• Wide Area Search • Position Updates • EO Sensor Cueing • In-Weather Target Designation

The Basic Radar Format

The basic radar format consists of symbology that is common between the RBM and HRM modes and operate in like fashion. Figure 1 shows the basic format of the APG-70 radar and highlights some of the symbology that is common between the two radar modes. In addition, the numbering of the bezel switches as can be seen from Figure 1 occurs in a counter-clockwise direction starting on the left side of the format under the BIT circle.

Common Radar Symbology

Radar symbology that are common between the RBM and the HRM modes that operate in like fashion are detailed in the following paragraphs. In addition, common symbology from the various mode of the APG-70 radar may also be found within the following paragraphs.

Azimuth and Elevation Scan Limits

The antenna azimuth and elevation scales provide a reference for determining the antenna position. The antenna elevation scale consists of nine horizontal indices, proportionally spaced to indicate 0, ±1, ±2, ±5, and ±10° elevation. The zero elevation mark is referenced to the horizon. The azimuth scale is used to determine the antenna position and consists of vertical indices, which correspond to 0° and ±60° antenna azimuth. The zero azimuth mark is referenced to the aircraft velocity vector (ground track). Both scales are in the same fixed location for all radar formats (e.g., RBM, HRM, PPI, BCN).

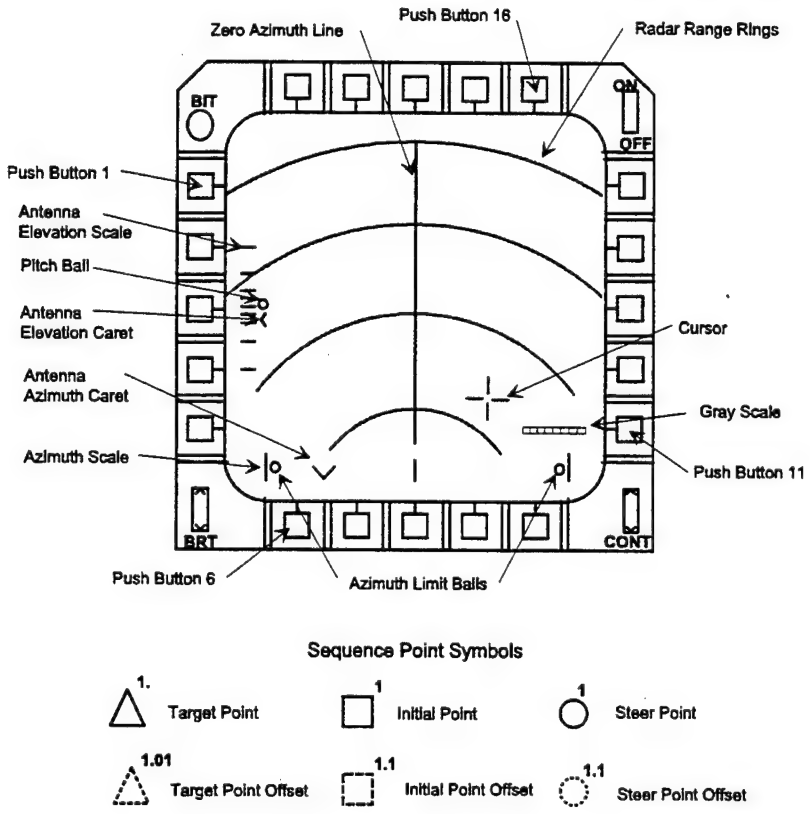


Figure 1. The Basic Radar Format

Antenna Carets

The antenna azimuth and elevation carets reflect the current antenna position on the format. The RBM, HRM PPI, and BCN modes are space stabilized.

Azimuth Limit Balls

The azimuth limit balls are positioned by the radar simulation and provide a reference for the azimuth scan limits. They are not constrained by the $\pm 60°$ antenna limits, but extend to the edge of the display format. They only display in the HRM mode. During HRM mapping, the balls indicate the position of the antenna scan relative to the blind zone and gimbal limits.

Pitch Ball

The pitch ball provides a reference of the current aircraft pitch angle with respect to the horizon. During steep climbs and dives, it is limited to the edge of the display format. The elevation is referenced to the aircraft body by observing the elevation caret position relative to the pitch ball position on the format. The radar changes the pitch ball symbol to reflect the source of the radar's attitude data.

Range Rings

Range arcs are provided on the RBM and HRM PPI mode formats, which represent 25, 50, 75, and 100% of the selected range.

Zero Azimuth Line

The zero azimuth line is displayed on the RBM format. The radar is drift stabilized up to $\pm 10°$ of drift.

PPI Gray Scale

A gray scale is provided displaying eight shades of gray embedded in the radar RBM and HRM PPI map video.

Cursor

The cursor is an open plus-shaped symbol that can be positioned by the crew or the central computer depending on the specific cursor function selected. The cursor is slewed using the TDC on the right throttle grip.

Sequence Point Symbols

If the declutter option is not selected, the sequence point symbols are displayed in the RBM and the HRM modes. The following paragraphs describe each point symbol.

- Steer Point - The steer point is a circle with a whole number assigned to each number (numbered 1 to 100).

- Steer Point Offset - The steer point offset is always associated with a steer point and represents a geographical point used to enhance off route sensor cueing and position updates. The steer point offset is a dashed circle with the number of the steer point it is associated with, followed by a decimal extension (e.g., 1.1, 2.1).

- Initial Point (IP) - The IP is a square with a whole number assigned to it (1 to 100). The IP indicates the last steer point prior to the target point.

- IP Offset - The IP offset is always associated with an initial point and represents a geographical point used to enhance off route sensor cueing and position updates. It is a dashed square with the number of the steer point it is associated with, followed by a decimal extension (e.g., 1.1, 2.1).

- Target Point - The target point is a triangle with a whole number identification assigned to it (1 to 100) and a decimal point (e.g., 1., 2.).

- Target Point Offset - The target point offset is always associated with a target point and it represents a geographical point used to enhance off route sensor cueing and position updates. The target point offset is a dashed triangle with the number of the target point it is associated with, followed by a two decimal extension (e.g., 1.01, 2.01).

Radar Mode Select

The APG-70 radar implementation provides a means, via push button 6, for selecting the three other radar modes (HRM PPI, BCN, and PVU) when in RBM. Successive depressions of the A/G radar mode push button steps through RBM, HRM PPI, BCN, and PVU in a rotary fashion. When the radar mode is selected, it is initialized at a range based on the aircraft altitude), as shown in Table 2. Altitude is computed from a radar altimeter system. If the radar altimeter is inoperative, the system altitude above the nearest sequence point is used. If both of these are invalid, the altitude above mean sea level is used.

Table 2. Aircraft Altitude

AIRCRAFT ALTITUDE	RADAR RANGE
At or below 500 ft.	10 nm
501 ft to 1000 ft	20 nm
Above 1000 ft	40 nm

Radar Range

The radar is capable of providing six range selections via push buttons 13 and 14: 4.7, 10, 20, 40, 80, and 160 natuical miles. In RBM, the selection of range is only available when the map cursor function is selected and the freeze function is not selected.

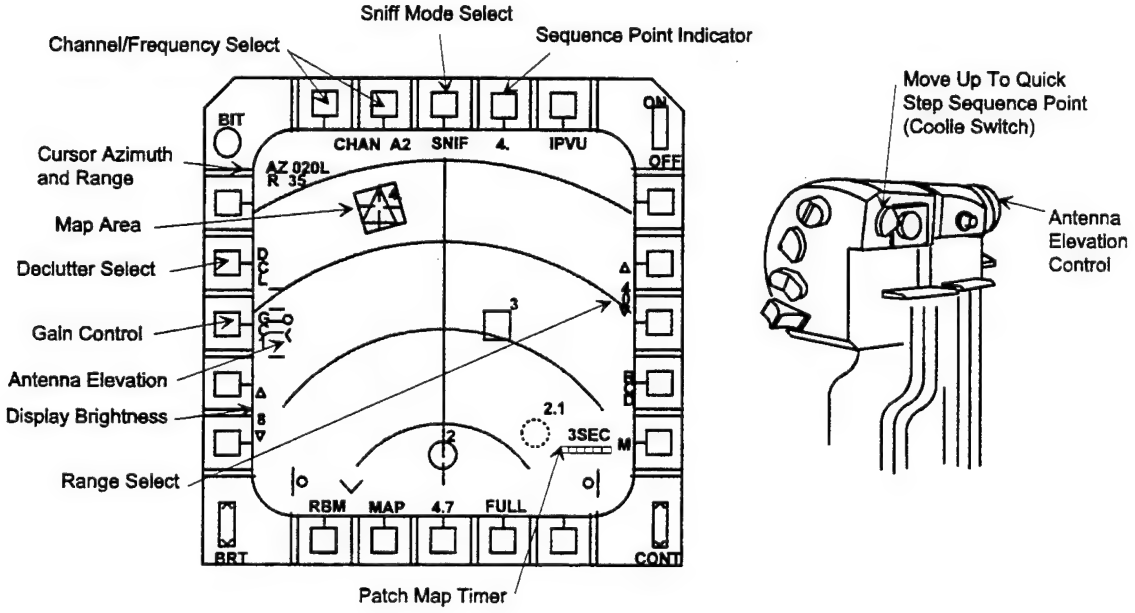


Figure 2. RBM Control Options

Sequence Point Indicator

The radar is capable of accessing stored sequence point (pre-planned geographical positions) data. The sequence point currently selected is displayed beneath push button 17, and the cursor is positioned over the sequence point, as shown in Figure 2. A maximum of five sequence points are displayed. If there are more then five sequence points within the selected range, the simulation only displays the five closest points to the aircraft. The currently selected sequence point can be changed using the up-front control entry, by depressing push button 17 or by moving the coolie switch on the throttle up (called quick stepping). Using any of these methods steps the cursor through the selected sequence points (including steer points, initial points, target points, and all offsets) displayed on the map in a rotary fashion (e.g., 1., 2., 2.1, 3., 4.). The cursor bearing and range information, which appear in the upper left corner of the display, is updated to reflect the current cursor position.

Frequency Band/Channel

The radar format provides the capability, via push buttons 19 and 20, for the pilot to change the radar frequency and channel. Push button 20 steps through bands A through E, and push button 19 steps through channels 1 through 8 in a rotary fashion. The selected frequency and channel are displayed to the pilot under the appropriate push buttons.

Declutter Option

The radar format provides the capability, via push button 2, for the pilot to remove the displayed sequence point symbols and associated sequence point numbers. When the declutter option is selected, the legend is boxed. Target points and target offsets are never removed, even when the declutter option is selected.

Freeze Mode

The radar format provides the capability, via HOTAS, for the pilot to freeze/unfreeze the radar map (see Figure 3). When the pilot selects freeze, continuous mapping is stopped following the completion of the current RBM display. When the radar is frozen in the RBM and the HRM is commanded, the radar constructs one HRM map and then freezes. The word FREEZE is displayed when the map is frozen to provide feedback to the pilot of the radar

status. The radar is returned to its operating mode by deselecting freeze in the same manner it was activated.

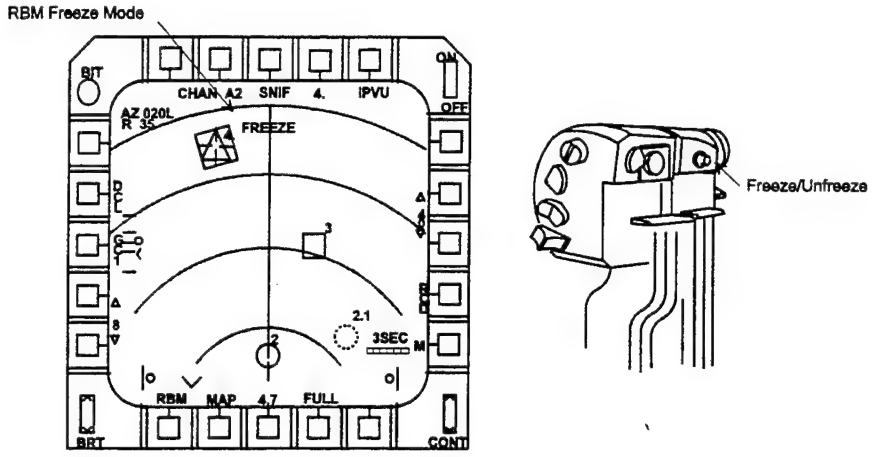


Figure 3. RBM Freeze Mode

Cursor Function

The radar format provides the capability to display and move a cursor (via the TDC on the right throttle) in the RBM mode, as shown in Figure 4. The cursor azimuth and range is presented in the upper left corner. It is capable of functioning in five different modes: map, update, target, cue, and mark. Successive depressions of push button 7 steps through MAP, UPDT, CUE, TGT and MARK in a rotary fashion.

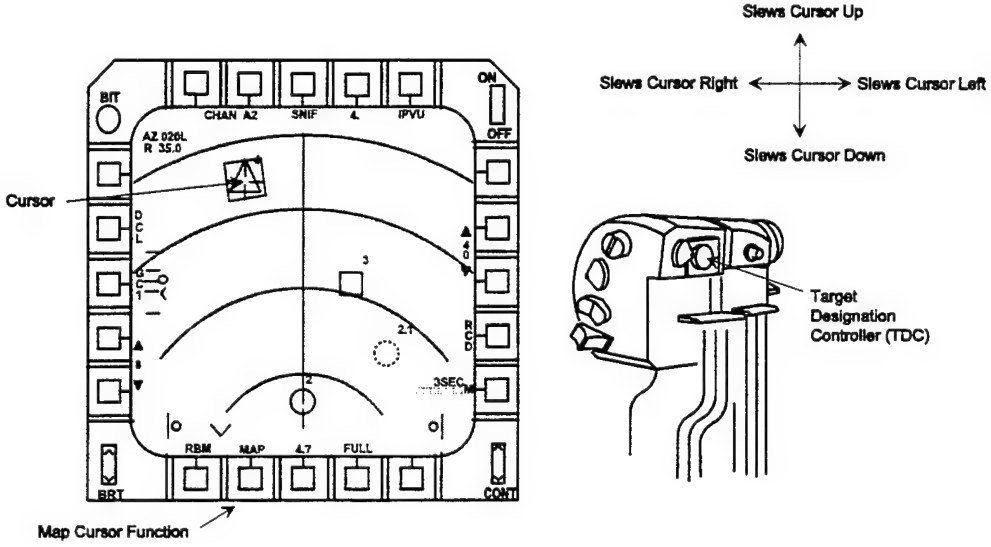


Figure 4. Map Cursor Function

- Map Cursor Function - The map cursor function is used to prepare the system for commanding HRM patch maps (see Figure 4). The map cursor function must be selected before an HRM can be commanded. The process for commanding a map is to select the map cursor function, select the appropriate display window size, quick step the cursor, or slew it with the throttle mounted TDC, over the area to be mapped, and depress and release the TDC.

- Update Cursor Function - The update cursor function must be selected (via push button 7) before the pilot can perform a position update of the Mission Navigator (MN) or the Inertial Navigation System (INS) from the RBM mode. When the update function is selected, it provides an update to the MN or INS with reference to the cursor position and the latitude/longitude of the selected sequence point. The capability to select either MN or INS for updating is provided by the radar mechanization and the update function is commanded via HOTAS (see Figure 5). The process for performing an update is to select the update cursor function, generate the error information by moving the cursor on the RBM format to a point on the map representing the actual location of the selected sequence point, and examine the position errors to determine if an update should be commanded. If an update is required, the pilot presses and releases the TDC. If less than or equal to 3000 feet, position errors are displayed in feet . If greater than 3000 feet the errors are displayed in nautical miles. When the INS is selected, all sequence points are displayed based on INS position, rather than MN position. During INS updates, a counter is displayed above the UPDT legend. It represents the number of seconds remaining until the RBM cursor position is too old to use for an INS update. Once the counter reaches zero, the legend INV is displayed in place of the update errors to indicate that the data is invalid. The cursor must then be repositioned to perform an INS position update.

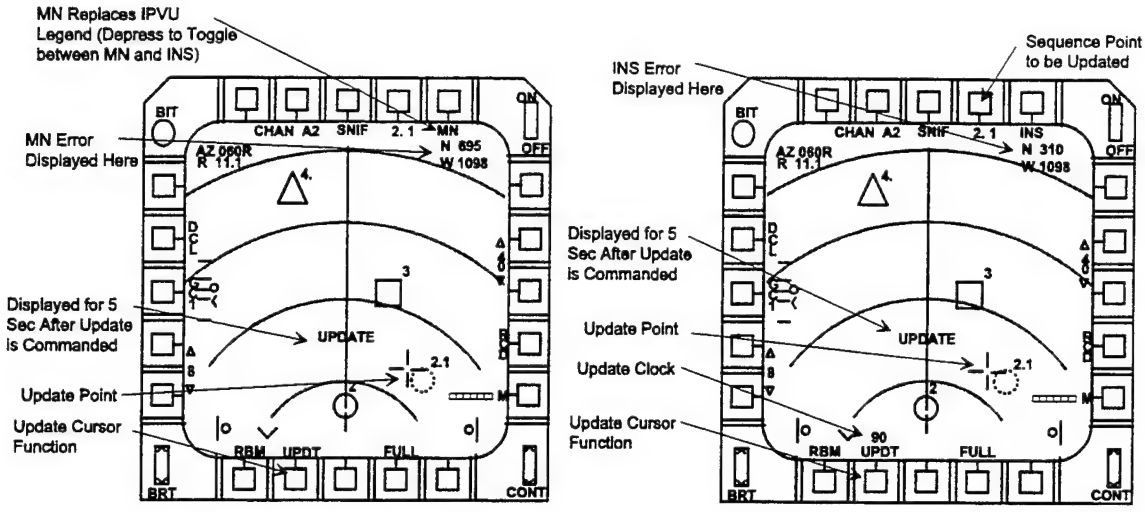


Figure 5. Update Cursor Function

- Cue Cursor Function - The cue function is used to direct or command a supporting imaging sensor (e.g., targeting FLIR) to a point on the RBM. When the cue function is selected (via push button 7, as shown in Figure 6) and the TDC is pressed and released, the sensor of interest is slaved to the point on the ground under the RBM cursor. Selecting the cue cursor function does not stop the RBM scan. The operator is also provided with a FREEZE capability to stop the continuous mapping.

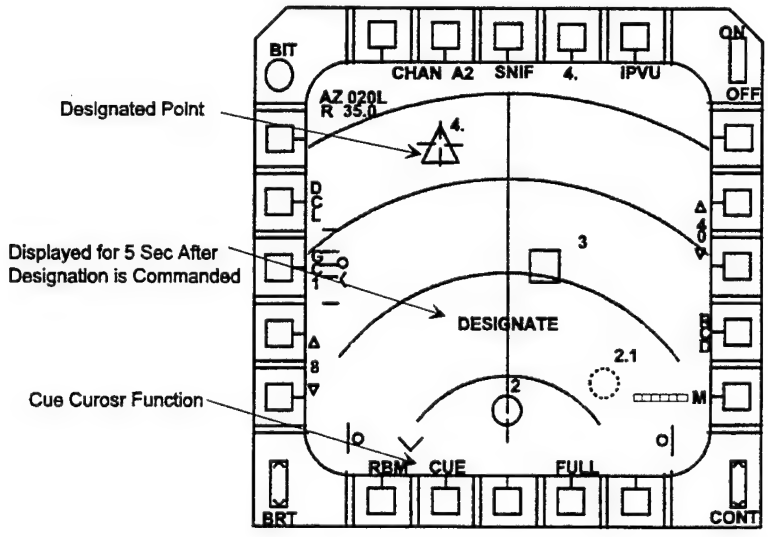


Figure 6. Cue Cursor Function

- Target - The target cursor function allows the operator to designate a point on the RBM as a target for weapon delivery. When the target cursor function is selected (via push button 7, as shown in Figure 7) and the TDC is pressed and released, the radar system designates the location of the cursor as a target or target offset, it cues the targeting infra-red (IR)/guided weapon sensor to that point and it provides ranging information to the computer for weapon delivery computations. Selecting the target cursor function does not stop the RBM scan. The operator is provided with a FREEZE capability to stop the continuous mapping. The target cursor function also enables the control of a pattern steering line for the designated target.

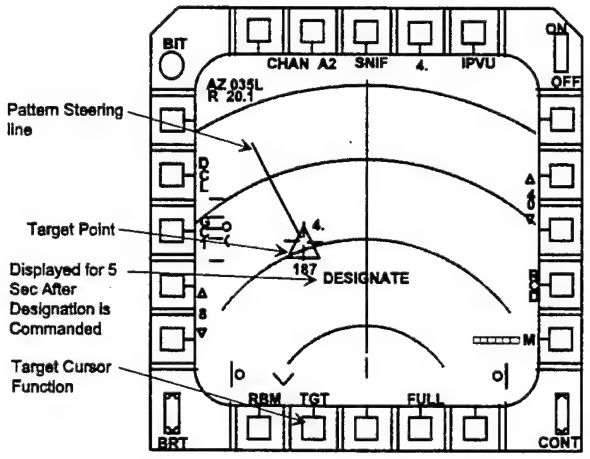


Figure 7. Target Cursor Function

- Mark - The mark cursor provides the pilot the capability to mark a specific point on the RBM for future reference. When the mark cursor function is selected (via push button 7, as shown in Figure 8) and the TDC is pressed and released, the radar system stores the latitude, longitude, and altitude of the cursor position and time of designation for later use.

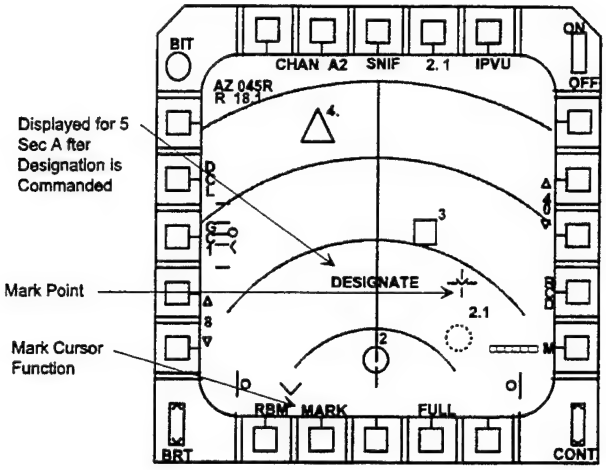


Figure 8. Mark Cursor Function

Display Window

The radar is capable of providing eight patch map sizes (also called display window sizes): .67, 1.3, 3.3, 4.7, 10, 20, 40, and 80 nm square. A patch map is a high resolution map of a designated area. See HRM section for a description of HRM patch mapping. Selection of the display window size is provided via HOTAS, as shown in Figure 9. When the HRM map is commanded, an outline of the encompassed area is displayed on the radar image. When a smaller size is commanded with .67 nm already selected, the display window indication blanks. When a map is commanded with the size blank, the map size is based on cursor position, as shown in Table 3.

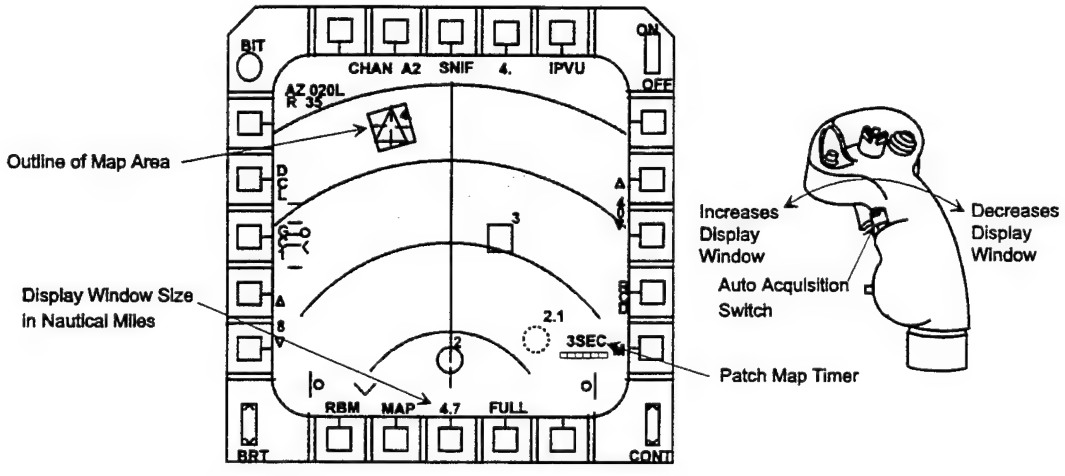


Figure 9. Display Window Control

Table 3. Cursor Position

CURSOR POSITION	DW SIZE
2.7 nm - 50 nm	3.3 nm
50 nm - 80 nm	4.7 nm
80 nm - 150 nm	10 nm

- Patch Map Timer - The radar simulation is capable of displaying to the pilot an estimate of how long it will take to construct an HRM patch map at the current cursor location, using the current display window size selected, and at the current aircraft geometry and speed (see Figure 9). The remaining estimated time is only displayed when the radar is in the map cursor function, when a display window size is selected, and when the cursor is over a mappable location.

Pattern Steering Line

The radar simulation provides a Pattern Steering Line (PSL) when a target has been designated. The PSL emanates from the designated target symbol (triangle) and indicates the target approach heading, as shown in Figure 10. Initially, the PSL indicates the current line-of-sight to the target as it is referenced to the RBM video. Additionally, the associated magnetic heading from the (A/C) to the target point is displayed under the target symbol. The simulation includes the capability for the pilot to position the PSL to a desired approach heading and then designates it. This provides a steering cue for the desired approach heading. To enable the PSL, slewing the pilot is required to pull aft on the auto acquisition switch while in the TGT cursor function. An arrowhead appears at the end of the PSL, and the PSL is controlled by the TDC. When the PSL is positioned to the desired run-in heading it is designated as the approach heading by pressing and releasing the TDC. The arrowhead then becomes a circle and the azimuth steering line on the HUD changes to a bank steering bar.

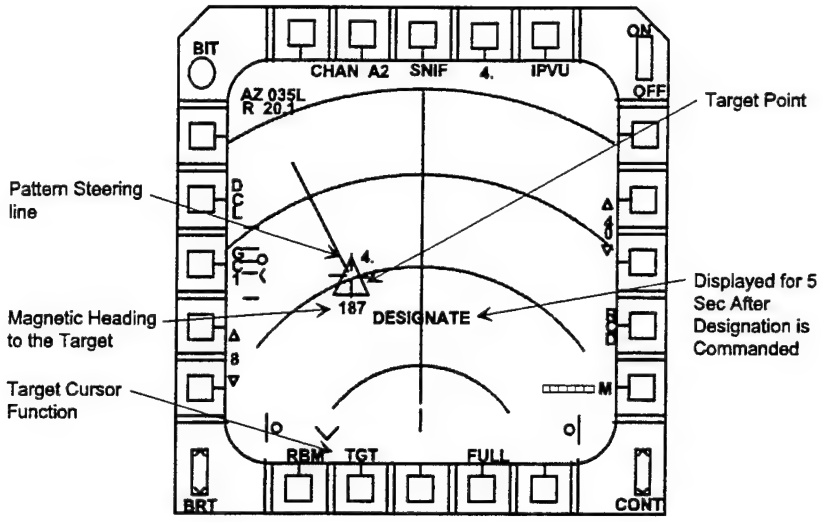


Figure 10. Pattern Steering Line Control

Interleaved Precision Velocity Update (IPVU)

The radar provides the capability, via push button 16, to select/deselect an automatic mode to update the MN (every 60 seconds), as shown in Figure 11. When the IPVU mode is selected, the legend is boxed and the velocity differences between the radar and the inertial navigation system are displayed and automatically accepted. Successive depressions of push button 16 toggles between the selected/deselected state.

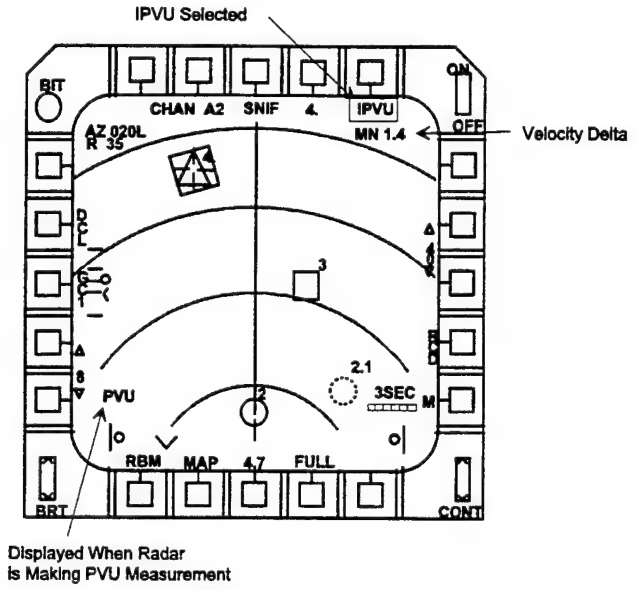


Figure 11. IPVU Format

Real Beam Map

The RBM mode is used to identify gross terrain features for navigation and weather detection and to cue the radar to a specific point for high resolution mapping, as presented in the Plan Position Indicator (PPI) format. This format is selectable from a main menu, as shown in Figure 12. The RBM mode takes approximately 1 second per sweep and provides six ranges: 4.7, 10, 20, 40, 80, and 160 NM. Also, the RBM incorporates the positionable cursor, sequence point data, and a means to transition to a high resolution map (called "patch map") presentation via hands-on-throttle-and -stick (HOTAS). The following information describes the modes of operation that are unique to the RBM.

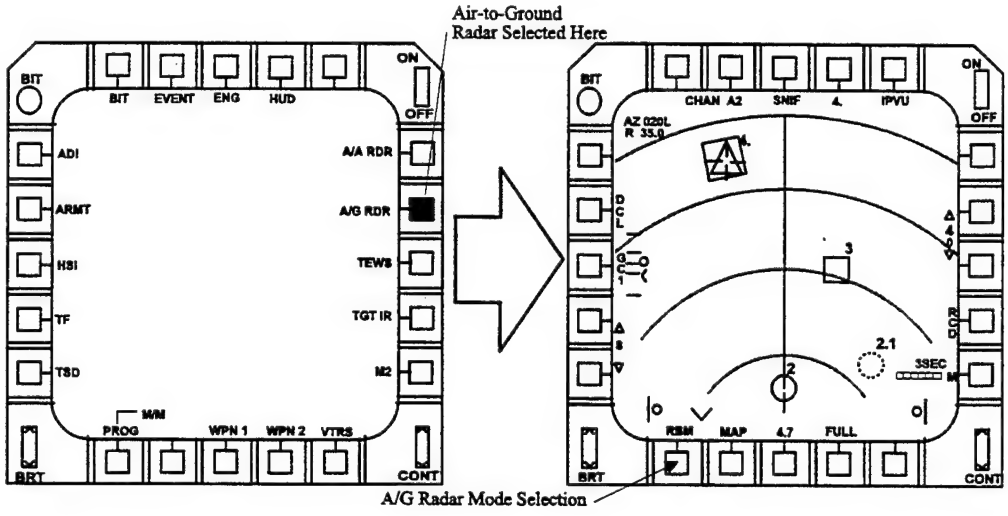


Figure 12. RBM Format Selected From Main Menu

SNIFF Mode

The radar implementation is capable of simulating operations in a "receive only" (SNIFF) mode to detect jamming of the radar channels or to provide a minimum radar radiation time to prevent detection. This mode is selected/deselected via push button 18. When the SNIFF mode is selected, the legend has a rectangular box drawn around it to indicate SNIFF has been selected.

Antenna Elevation

The radar controls provide the capability, via HOTAS, to position the radar's antenna in elevation. An indication of the antennas elevation is displayed to the pilot.

Display Brightness

The radar controls provides the capability, via push buttons 4 and 5, for the pilot to adjust the brightness of the radar display over 16 (0-15) selectable settings. Successive depressions of the display brightness push button steps through the settings in a rotary fashion. The selected brightness level is displayed to the pilot.

Antenna Azimuth Scan

The radar is capable of providing three selectable scan widths: full, half, and quarter. The selection of the scan width is provided via push button 9, as shown in Figure 13. Successive depressions of this push button steps through full, half, and quarter in a rotary fashion. Each scan is centered on the aircraft velocity vector. In the FREEZE mode, azimuth scan changes are ignored.

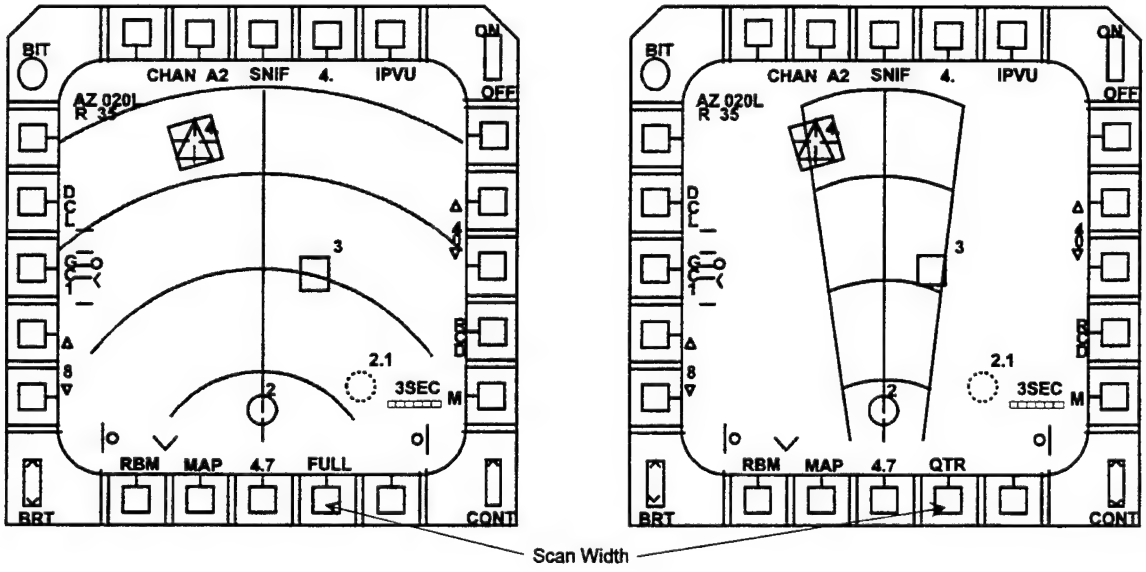


Figure 13. Antenna Azimuth Scan Select

High Resolution Map

The HRM mode provides the capability to produce high range and azimuth resolutions utilizing synthetic aperture radar (SAR) mapping techniques. It provides the capability to produce both the Plan Position Indicator (PPI) format and a "patch map" format (see Figure 14). The HRM PPI format is similar to the RBM PPI format, except it does not provide a real-time radar picture due to the delay in Doppler processing. Also, the HRM PPI format does not provide radar returns for a $\pm$ 8 deg. area, called the "blind zone," around the aircraft ground track due to the Doppler notch. HRM ranges are the same as in RBM. In the PPI mode, the radar takes up to 18 seconds per sweep (vice 1 second for RBM).

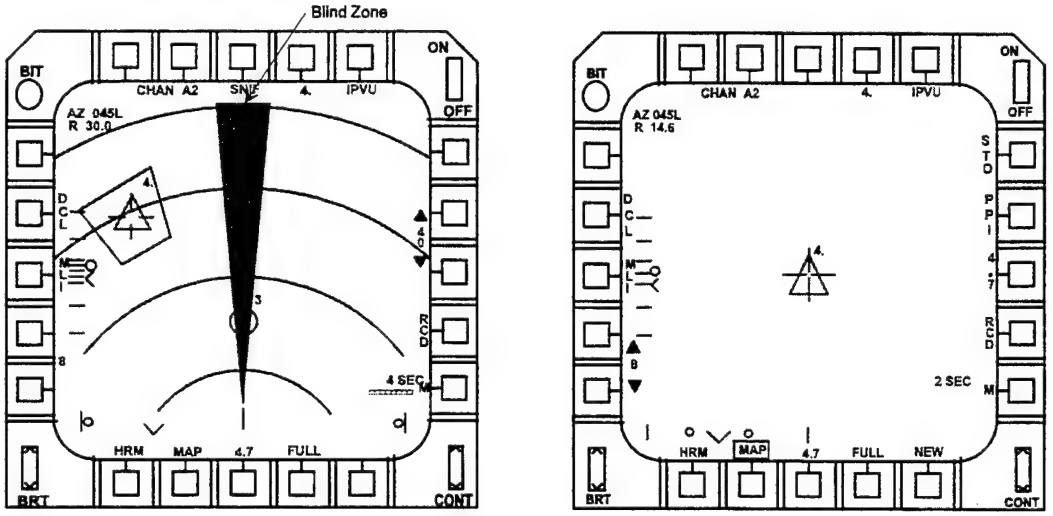


Figure 14. HRM PPI and HRM Patch Map Formats

Patch maps can be commanded from RBM, HRM PPI, or another patch map any time the cursor function is in "MAP." The process for making a patch map is to select the MAP cursor function at push button 7, position the cursor over the area to be mapped, select the desired display window size using the auto acquisition switch on the throttle, ensure that the mapped area is not within the 8 deg blind zone and then depress and release the throttle mounted Target Designation Controller (TDC). The parameters for the required minimum and maximum ranges, which are based on display window sizes, are listed in Table 4.

Table 4. Minimum and Maximum Ranges Based On Display Window Size HRM Patch Map Parameters

Display Window (NM)	Min/Max Mapping Range (NM)
0.67	4.7 / 20
1.3	4.7 / 40
3.3	4.7 / 50
4.7	4.7 / 80
10	10 / 160
20	20 / 160
40	40 / 160
80	80 / 160

Descriptions of the modes of operation that are unique to HRM are provided in the following paragraphs.

Antenna Azimuth Scan

Unlike the RBM, the HRM PPI scan is centered on the radar cursor (not the velocity vector as in RBM). The cursor is capable of slewing the scan center to a position within the displayed range (see Figure 15). In half scan, the scan center is capable of being moved up to 25° either side of the velocity vector. In quarter scan, the center is capable of being moved up to 37.5°.

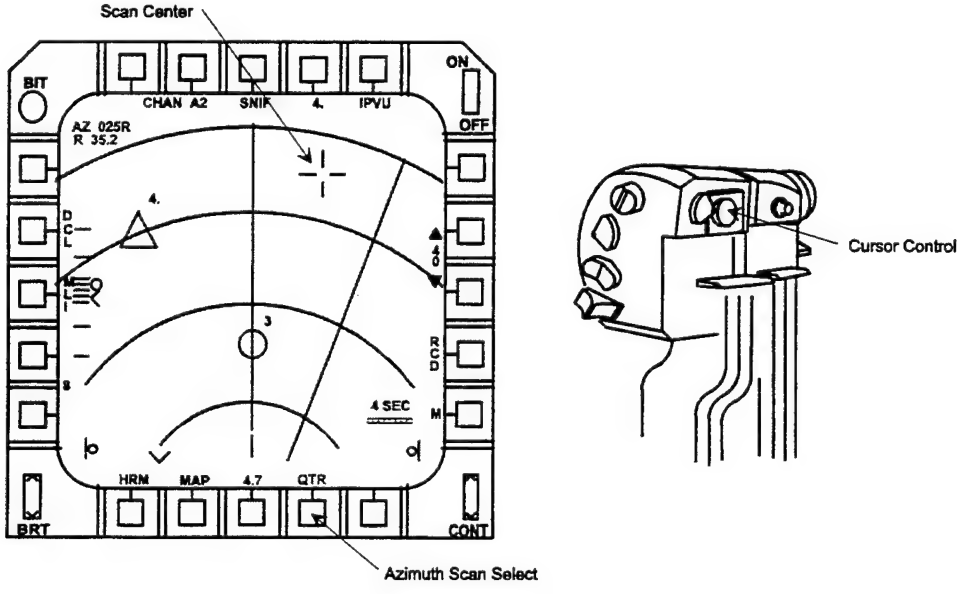


Figure 15. Azimuth Scan Format

SNIFF Mode

The SNIFF mode is the same as in the RBM, except that it causes the radar to cease transmissions immediately and enters into a freeze condition.

Antenna Elevation

The antenna elevation is automatically centered on the cursor position and is not adjustable.

Multi-Look

The radar format provides the capability to select, via display push button, two multi-look options. These options vary the video processing and take the place of the gain function in the RBM (see Figure 16). Two multi-look options are selectable (ML1 and ML2) through successive depressions of push button 3. When ML2 is selected, the video quality is improved, but the map processing time increases by approximately 1.5 times.

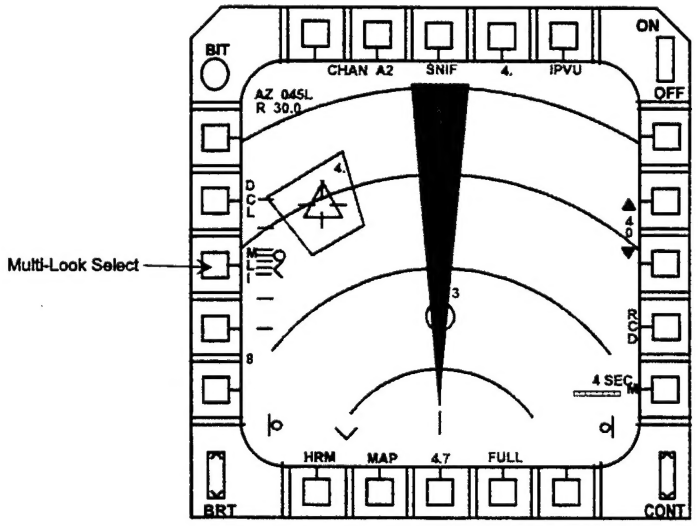


Figure 16. Multi-Look Option

Display Brightness

The display brightness in the HRM functions the same as in the RBM, except that it is only available in the HRM freeze.

Set Function

The radar provides the capability, via display push button, for the pilot to continue to perform HRM targeting functions when the radar has been commanded to air-to-ground (A/G) ranging or to an air-to-air mode (see Figure 17). Whenever the radar is changed to either air-to-air (A/A) or A/G ranging from HRM, the last HRM map is automatically frozen on the A/G radar format and the SET legend is displayed below the top center push button. Selecting the SET push button causes a box to be placed around the legend and provides the capability to perform HRM cursor designations without interrupting present radar operations. All cursor functions, except map, are then available. The declutter, azimuth scan select, map recall, store, and IPVU push button options are not functional in SET, and the A/G radar mode is not selectable until the set mode is exited. To exit the set mode, the operator must deselect the SET option via the display push button.

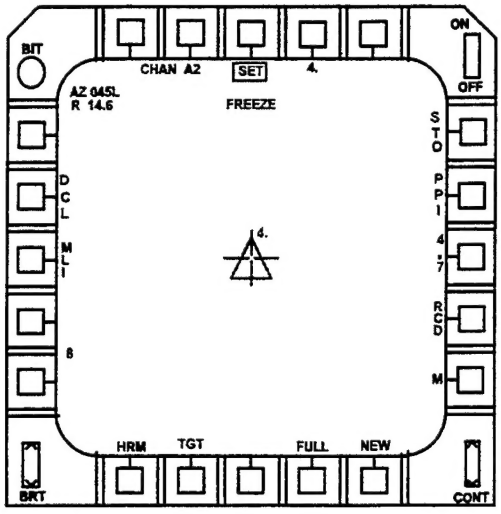


Figure 17. Set Mode Format

HRM Patch Map Modes

When an HRM patch map is commanded the following submodes are available:

- Map Stabilizing - The radar is capable of providing two map stabilizing options via display push button: progressive and stabilized. Successive depressions toggles between the two options (see Figures 18 and 20). When progressive is selected, the azimuth and range from the aircraft to the area being mapped remains the same. When stabilized is

selected, the point on the ground under the cursor is continuously mapped.

- PPI - The radar simulation provides the capability, via push button 14, for the pilot to select the PPI mode once a patch map is commanded (see Figure 18). Selecting the PPI mode halts the patch map and begins HRM PPI mapping.

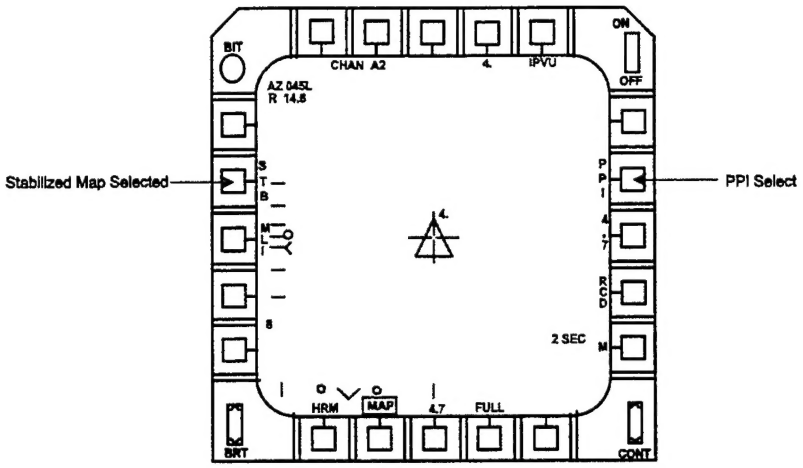


Figure 18. HRM Patch Map Mode Options

- Recall - When the freeze mode is selected in HRM, the radar simulation provides the capability, via display push button, for the pilot to flip back and forth between the most recently commanded patch map and the second most recently commanded patch map (see Figure 19). When freeze is selected, the lower right push button reads NEW to indicate that the displayed map is the one most recently created. Pressing this push button with NEW displayed, selects the OLD or prior map from memory. Subsequent depressions of the push button toggles between the two in a rotary fashion.

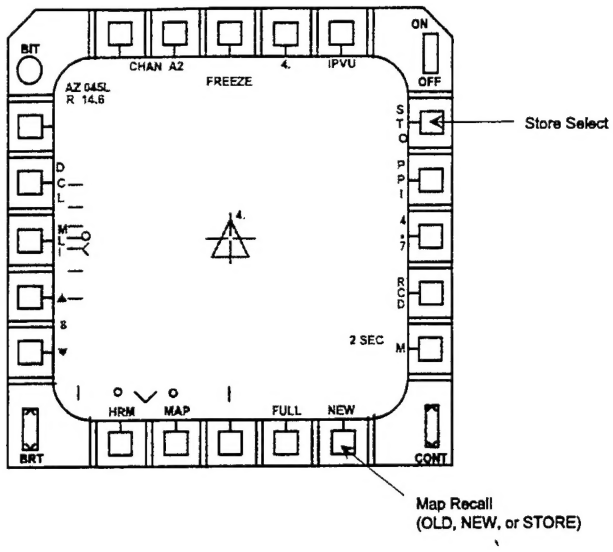


Figure 19. HRM Store Mode Option

- Store - The radar simulation provides the capability, via push button 15, for the pilot to store a frozen patch map (see Figure 19). Once the STO option is selected, the radar saves the current map and stores it in memory. The store selection is only available in the freeze mode. With the store option is selected, the recall push button legend is changed to STORE indicating that the currently displayed map is protected. Subsequent depressions of the push button toggles between the stored map and the most recently created map in a rotary fashion.

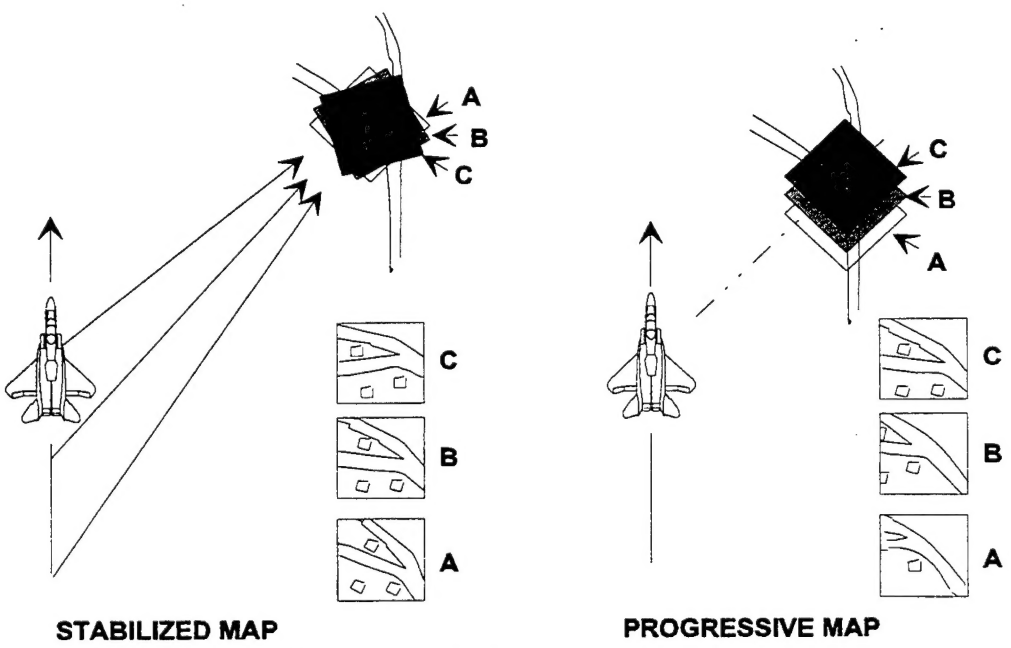


Figure 20. HRM Mapping Options